4

Statistics

4.1 The Subset Sum Problem (Part 1)

The subset sum problem (SSP) involves deciding whether any subset of the integers in a set S sum precisely to T. For example, if $S = \{5, 6, 2, 3\}$ and $T = 5$, then the answer is 'yes' since one possible subset is $\{2, 3\}$.

We'll consider a variation of this problem, where we want all the subsets with a sum $\leq T$. For instance, using the previous S and T, the subsets are $\{\{\}, \{2\}, \{3\}, \{5\}, \{2, 3\}\}$. We're focusing on this version since it'll be useful for several examples in this chapter.

Listing 4.1 (sumSubsets.py) is a recursive solution, which is both relatively easy to understand and also suffers from exponential running time as the set size increases.

```
def sumSubsetsR(nums, tot):
  combs = sumSubsets(nums, tot, 0, [])
  return sorted(combs, key=sum)

def sumSubsets(nums, tot, idx, comb):
  sumC = sum(comb)
  if sumC == tot:
    return [comb]
  elif sumC > tot:
    return []
  else:
    combs = [comb]
    for i in range(idx, len(nums)):
      combs.extend(sumSubsets(nums,tot,i+1,comb+[nums[i]]))
    return combs
```

Listing 4.1. A recursive subset sum

A typical call would be:

```
>>> from sumSubsets import *
>>> sumSubsetsR([5, 6, 2, 3], 5)
[[], [2], [3], [5], [2, 3]]
```

Note that although this is a set problem, we've decided to code things using lists. This allows us to include multiple copies of a value in S if necessary.

The crucial lines in sumSubsets() are its else-branch which builds a list of 'subsets' containing both the current list (comb) and every version where one or more numbers have been added by recursively calling sumSubsets().

As we've seen before, the standard way to speed up recursive code is to utilize memoization data structures (typically lists) to record subcalculations so they don't need to be repeated. sumSubsetsD() in sumSubsets.py maintains a Python dictionary whose key/value pairs are a sum and a list of lists that add up to that sum. This allows the code to be rewritten to be iterative, with a running time of $O(n^2)$, but with a potentially large memory overhead for the dictionaries. Such functions are usually classified as having *pseudo-polynomial* running time since the big-Oh calculation doesn't capture all the costs related to the problem size.

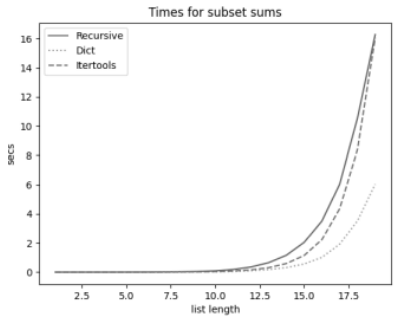


Figure 4.1. Subsets sum running times

Our last subset-sum solution utilizes Python's `itertools.combinations(seq, t)` which generates t-length subsequences of elements from seq. This is almost what we want, but we also have to generate all the smaller subsequences for $0 \leq x < t$, as shown in Listing 4.2 (sumSubsets.py).

```
def sumSubsetsC(nums, tot):
  combs = [list(comb) for r in range(len(nums)+1)
                for comb in itertools.combinations(nums, r)
                   if sum(comb) <= tot]
  return sorted(combs, key=sum)
```

Listing 4.2. An itertools subset sum

subsTimer.py uses the timeit module to time these three functions when given a list of 1's (e.g. [1], [1,1], [1, 1, 1], up to 20 1's) and tasked with generating

all the sublists that sum to 8 or less. The plot of their running times is shown in Fig. 4.1.

The most surprising thing about Fig. 4.1 is that Python's itertools.combinations() has a similar exponential running time as sumSubsetsR(), but this may be due to our use-case, which calls it $t + 1$ times for each list.

4.1.1 The Number of Subset Sums. One way to speed up our code is to simplify the coding problem! Often the actual sublists aren't required, just the total number. For instance:

```
>>> from countSums import *
>>> countSumsR([5, 6, 2, 3], 5)
5
```

Listing 4.3 (countSums.py) is recursive, with three boundary cases and an else-branch expressing the idea that the number of subsets involves adding together two subtotals – one where the current number in the list is not used, and the other where it is (which means deducting it from the required total).

```
def countSumsR(nums, tot):
  return countSums(nums, tot, len(nums))

def countSums(nums, tot, n):
  if tot < 0:
    return 0
  elif (tot == 0) or (n == 0):
    return 1
  else:
    return countSums(nums, tot, n-1) + \
           countSums(nums, tot-nums[n-1], n-1)
```

Listing 4.3. A recursive subset sum counter

In the same way as before, we can augment recursion with memoization, b this time we can utilize a 2D list which has a fixed size (unlike the dictionary in sumSubsetsD()).

Listing 4.4 (countSums.py) again illustrates that memoization usually permits recursion to be replaced by iteration, which in this case fills the table row-by-row.

```
def countSums2D(nums, tot):
  n = len(nums)
  tots = [[0 for j in range(tot+1)] for i in range(n+1)]
  for j in range(tot+1):     # set first row
    tots[0][j] = 1
  for i in range(1, n+1):    # set d's for other rows
    d = nums[i-1]
    for j in range(d):
      tots[i][j] = tots[i-1][j]
```

```
    for j in range(d,tot+1):
      tots[i][j] = tots[i-1][j] + tots[i-1][j-d]
  return tots[n][tot]
```

Listing 4.4. An iterative 2D subset sum counter

A close look at how the table is filled shows that the code only utilizes the current and previous rows, so the table could easily be made smaller. A further trick is to notice that the current row is filled left-to-right, so we only actually need to store one row. These observations result in Listing 4.5 (countSums.py).

```
def countSums1D(nums, tot):
  n = len(nums)
  tots = [1]*(tot+1)
  for i in range(1,n+1):
    d = nums[i-1]
    for j in range(tot, d-1, -1):
      tots[j] += tots[j-d]
  return tots[tot]
```

Listing 4.5. An iterative 1D subset sum counter

And so we conclude with a function that has linear running time, although we haven't actually quite finished yet – we'll return to this topic one more time in Sec. 5.9.

4.2 Matched Pairs

We'll study the simple but important problem of matched pairs by considering three examples.

4.2.1 Marijuana vs. Tobacco. The data in Table 4.1 comes from a study where nine individuals had their mental performance tested 15 minutes after smoking an ordinary cigarette (row X) and 15 minutes after consuming a marijuana cigarette (row Y), with positive numbers denoting improvements. To deal with the possible effect of smoking different substances in a certain order, a flip of a coin for each subject decided which type of cigarette they smoked first. The D row measures how much more positive the effects of marijuana were compared to tobacco.

The table shows that different people experienced different effects, but can we nevertheless infer something about the effects on the group, or are the fluctuations in mental performance just random? To deal with such a possibility, we formulate two hypotheses. The first is:

> **The null hypothesis** H: The fluctuations shown in Table 4.1 *are* just random. There's no difference in the effects produced by the two types of cigarette.

X	-1	-1	-3	3	-3	-3	2	4	10	$\sum X = 8$
Y	1	-3	-7	-3	-9	5	-6	-7	-17	$\sum Y = -46$
D = Y-X	2	-2	-4	-6	-6	8	-8	-11	-27	$\sum D = -54$
$\lvert D\rvert = d_k$	2	2	4	6	6	8	8	11	27	$sum\lvert D\rvert = 74$

Table 4.1. Marijuana study. X = tobacco; Y = marijuana. Source: *Science* 162, 1234-1242.

However, the experimenters, based on anecdotal evidence and the many negative differences in the D row, suspected that marijuana had a more detrimental effect than ordinary cigarettes. So they decided to test:

> **The alternative** A: Mental performance is *lower* after smoking marijuana than after smoking tobacco.

Hypothesis H can be given a precise meaning by making it equivalent to assigning $\pm$ sign differences by tossing a fair coin. Thus, with n people we would generate 2^n equiprobable cases.

Next, we choose a *test statistic* T which expresses quantitatively the amount of evidence for A contained in the differences D. We always define T so that a smaller value means that A is more likely to be true. Suppose the observed value of T is t. We must compute

$$P = P(T \le t|H) = \text{probability that } T \le t \text{ if } H \text{ is true.}$$

P is the *observed significance level*, or *P-value*, of the test, and the smaller the P-value the stronger the evidence for A.

We set $T = \#$ of positive signs in the D row, since as the number of positives decreases so A becomes more likely to be true (i.e. that marijuana is detrimental).

The observed value of T is $t = 2$ for a sample size of 9. There are $\binom{9}{0}+\binom{9}{1}+\binom{9}{2}$ $= 1 + 9 + 36 = 46$ cases which are favorable to event T ≤ 2. Thus,

$$P = P(T \le 2|H) = 46/2^9 = 9\%$$

Most statisticians only consider a P-value to be significant if it's 5% or less (and often less than 1% for small sample sizes, as here). The other, more fundamental problem is that this sign test doesn't make good use of the data. We should try to capture the fact that the few positive differences in the D row are much smaller on average than the negative differences.

A better statistic would be to measure the *size* of the positive differences vs. random differences for the nine subjects. This means that the null hypothesis would now be that all 2^9 differences are equally possible.

A quick look in row D identifies $T \le 10$, the sum of the positive differences. This is converted into the number of cases that give a total score of ≤ 10 based

on the sample data. For this we'll use the $|D|$ row since we're ignoring sign, and can employ subsetSumD() to find all the subsets which sum to 10 or less.

```
>>> sumSubsetsD([2,2,4,6,6,8,8,11,27], 10)
[[], [2], [2], [2, 2], [4], [2, 4], [2, 4], [6], [6], [2, 2, 4], [2, 6],
[2, 6], [2,6], [2, 6], [8], [8], [2, 2, 6], [4, 6], [2, 2, 6], [4, 6],
[2, 8], [2, 8], [2, 8], [2, 8]]
>>> len( sumSubsetsD([2,2,4,6,6,8,8,11,27], 10))
24
```

There are 24 ways to get positive scores of 10 or less. Thus

$$P(T \leq 10|H) = \frac{24}{512} = \frac{3}{64} = 4.6875\%.$$

This P-value is technically significant, but should really be taken as a sign to redo the experiment with a larger sample size.

What we have just done manually, and via the Python prompt, can of course be more fully automated. `mpairs.py` loads a text file containing matching pair data (e.g. see `marijuana.txt`), and analyzes the table to create a call to sumSubsetsD() like the one above. This program can also be used to calculate the results in the next example.

4.2.2 IQ vs. Weight. We would like to investigate the mental development of identical twins in relation to their weights at birth. The IQs of 12 pairs were measured years later and compared with their weights at birth. The X row in Table 4.2 holds the IQs of the heavier twin while the Y row are for the IQs of the lighter twin.

X	100	124	108	91	100	91	79	80	95	104	100	119
Y	101	123	106	97	106	84	70	70	84	92	85	104
D = Y - X	1	-1	-2	6	6	-7	-9	-10	-11	-12	-15	-15
$\|D\| = d_k$	1	1	2	6	6	7	9	10	11	12	15	15

Table 4.2. IQs of twins. X = heavy; Y = light. Source: *Child Development*, vol. 38, 623-629.

Our hypotheses:

- H: The heavy and the light twins develop the same IQ.
- A: The heavy twin usually develops a higher IQ.

Recall that we must define T so that a smaller value means that A is more likely to be true. We do that by summing the positive values in the D row of Table 4.2 which occur when the lighter twin has a higher IQ. The sum is $T = 1+6+6 = 13$.

Under H all 2^{12} (4096) subsets of the differences have the same probability, and the favorable cases for A are those subsets of d_k with sum to $T \leq 13$. We again find these by calling sumSubsetsD():

```
>>> sumSubsetsD([1,1,2,6,6,7,9,10,11,12,15,15], 13)
[[], [1], [1], [1, 1], [2], [1, 2], [1, 2], [1, 1, 2], [6], [6],
[1, 6], [1, 6], [1,6], [1, 6], [7], [1, 1, 6], [2, 6], [1, 1, 6],
  : # more lines, not shown
[1, 11], [12], [1, 6, 6], [1, 6, 6], [6, 7], [6, 7], [1,1, 2, 9],
[1, 2, 10], [1, 2, 10], [1, 1,11], [2, 11], [1, 12], [1, 12]]
>>> len(sumSubsetsD([1,1,2,6,6,7,9,10,11,12,15,15], 13))
60
```

There are 4096 possible cases, and 60 cases favorable for T $\leq$ 13. Thus

$$P = P(T \leq 13|H) = \frac{60}{4096} = \frac{15}{1024} = 1.465\%.$$

This P-value is strong evidence for A – a better nourished twin develops a higher IQ.

4.2.3 Cross vs. Self Fertilization. Finally, we consider a classic experiment by Charles Darwin which appears in his *The Effects of Cross and Self Fertilisation in the Vegetable Kingdom* (`https://www.gutenberg.org/ebooks/4346`).

He took 15 pairs of seeds of the same plant and planted them in 15 pots. One seed of each pair was produced by cross fertilization, the other by self fertilization. For pot #i he measured the height x_i of the cross fertilized plant and the height y_i of the self fertilized plant. For the difference $z_i = x_i - y_i$ he got (in 1/8-ths of an inch):

$$6, 8, 14, 16, 23, 24, 28, 29, 41, -48, 49, 56, 60, -67, 75.$$

Our hypotheses:

- H: There is no difference between the two kinds of seed.
- A: Cross fertilized seeds develop taller plants.

If H is true then we have 2^{15} possible and equiprobable cases.

Once again we define T so that a smaller value means that A is more likely to be true. We sum the negative values in the data row, which indicate when the self fertilized plants are taller than the cross fertilized ones. The sum of the negative differences is $t = 115$ (|-48 + -67|).

In the two previous examples, we used sumSubsetsD() at this stage, but these types of problem don't need the subsets only their total number. This allows us to switch to countSum1D(), which would actually be essential if the problem's sample size was bigger.

```
>>> from countSums import *
>>> countSums1D([6,8,14,16,23,24,28,29,41,48,49,56,60,67,75], 115)
863
```

We can now calculate P:

$$P = \frac{863}{2^{15}} = \frac{863}{32768} = 2.63\%.$$

This suggests that there is significant evidence for A: cross fertilized seeds do indeed develop into taller plants.

4.3 P-value by Simulation

Let's return to the marijuana example. If we plot Y versus X (Fig. 4.2) with `plotPairs.py` we see that there seems to be little relationship between the performances after smoking the two types of cigarette. This suggests that's there not much to be gained by considering the data in pairs.

Another approach is to treat the data (-3, 5, 10, -17, -3, -7, 3, -3, 4, -7, -3, -9, 2, -6, -1, 1, -1, -3) as random selections from an 'Effects of smoking on performance' probability spinner. The observed subset X due to tobacco smoking has sum $S = \sum X = 8$, while the subset Y due to marijuana smoking has sum $T = \sum Y = -46$.

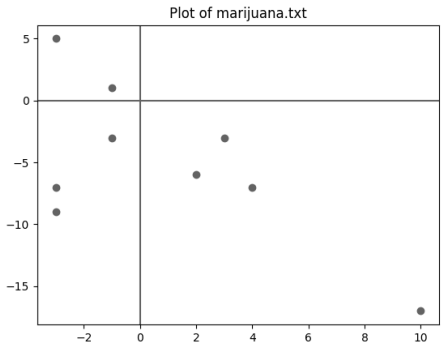


Figure 4.2. Performance affected by smoking (X=tobacco; Y=marijuana)

Our hypotheses:

- H: The large difference in the two sums is due to chance.
- A: The marijuana subset has lower performance than the tobacco subset.

If H is true, then X (the tobacco sample) is just a random 9-subset of the 18 data samples. So the question arises: how likely is it that a randomly chosen 9-subset of the data has a sum $S \geq 8$?

We can answer that through simulation – repeatedly draw a random 9-sample from the data and count how many times it sums to ≥ 8.

`permSim.py` utilizes code from Listing 3.10, packaged up as a sample() function.

To save some typing the tobacco and marijuana data is read from `marijuana.txt`:

```
2 9
-1 -1 -3  3 -3 -3  2  4  10
 1 -3 -7 -3 -9  5 -6 -7 -17
```

The simulation runs for NUM_TRIALS (32000) iterations, and reports the following:

```
> python permSim.py
fnm? marijuana.txt
x, y sums: 8 -46
count == 538; P == 0.016813
```

Over multiple runs, $P = P(S \geq 8|H)$ seems to vary between 1.5% and 1.7%.

4.3.0.1 Median and Confidence Interval. It's possible to extract more statistics from the simulation by analyzing the spread of P-values returned by multiple calls to permSim.py.

permSimStats.py generates 24 P-values, which it summarizes in a a stem-and-leaf plot. From these numbers we can extract an error estimate for the median P-value.

```
> python permSimStats.py
fnm? marijuana.txt
x, y sums: 8 -46
[1.69375, 1.74375, 1.50625, 1.740625, 1.6, 1.615625, 1.49375,
1.653125, 1.60625, 1.5875, 1.496875, 1.65625, 1.684375, 1.60625,
1.653125, 1.465625, 1.490625, 1.590625, 1.659375, 1.50625,
1.659375, 1.575, 1.484375, 1.6]
stem   leaf
   0 |
   1 | 444445555566666666666677
   2 |
```

We throw away the lower and upper quarters of the stem-and-leaf data. What remains is the *interquartile range* IQR $= 1.6 - 1.5 = 0.1$, and the median $C = (1.6 + 1.6)/2 = 1.6$.

McGill, Tukey and Larsen, in their 'Variations of Box Plots', *The American Statistician*, Vol. 32, No. 1 (Feb., 1978), pp. 12-16, developed a rule of thumb for estimating the true median with a 95% confidence interval:

$$C \pm \frac{1.57\ \text{IQR}}{\sqrt{N}} = 16 \pm (1.57 * 0.1)/\sqrt{24} \approx 1.6 \pm 0.03.$$

4.4 P-value by Computation

Let's consider the marijuana example one last time. The Null hypothesis is that it makes no difference what kind of cigarette a person smokes. If this were so then all 18 data samples could be regarded as values of the same random variable. In that case, the subset of 9 values for tobacco (the X data) would be just a randomly picked subset of 9 out of the 18. However, the sum of these 9 values is 8, which

seems rather large. To decide how likely it is that this happens just by chance, let's check all $\binom{18}{9}$ or 48620 9-element subsets of the smoking data set:

$$(-3, 5, 10, -17, -3, -7, 3, -3, 4, -7, -3, -9, 2, -6, -1, 1, -1, -3).$$

permCalc.py traverses the 9-subsets in lexicographic order and counts those which sum to 8 or more.

How do we generate all k-subsets of an n-set {1, 2, ..., n}? We start with {1, 2, ..., k}, and the next subset is found by scanning the current one from right to left to locate the rightmost index that has not yet reached its maximum. This element is incremented by one, and all positions to its right are reset to their lowest values.

In the following call, k is the size of the subsets, while tot is the sum that must be equaled or exceeded.

```
> python permCalc.py
k tot=? 9 8
count == 766
```

Exactly 766 9-subsets have a sum of 8 or more, so we have a P-value = 766/48620 = 1.5755%. This is the "exact" significance level for this data, and will be the value that permSim.py converges upon as the number of repetitions are increased towards ∞.

permCalc.py's traversals of the subsets has exponential running time, so is really only useful for small search spaces. For example, the number of 18-subsets of a 36-set is 9075135300. However, large sizes can be estimated using the approximation

$$\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$$

which determines $\binom{36}{18}$ to be around 9138368253.

In Sec. 4.6 we'll deal with this subset counting task more efficiently. Alternatively, for very large problems, we could use permSim.py from the preceding section. It gives practically the same significance level. For instance, permCalc.py reports the P-value to be 1.5755% while permSim.py estimates it to be around 1.6%.

Exercises for Sections 4.1 to 4.4

(1) Should hospitals allow parents to stay with their sick children (the rooming-in model) to shorten their stay? 25 pairs of children were considered, with relevant variables nearly the same in all cases, except that the parents of one child in each pair were allowed to stay at the hospital. Table 4.3 shows the number of days each child stayed in the hospital; X was the no-parents group, Y the rooming-in group.

Find D = X - Y and compute the P-value with `countSums.py`. The data in Table 4.3 can be found in sickDays.txt.

X	29	44	18	15	15	7	15	11	12	12	7	11	7	22	7	14	28	31	18	16	10	11	29	8	23
Y	29	32	16	8	8	7	14	8	7	12	7	8	7	17	6	19	12	8	8	6	10	8	21	17	15

Table 4.3. Days in hospital. Source: Fernandez-Jung, F., Dissertation FU Berlin, 1983.

(2) Use `permSim.py` to analyze the data in Table 4.3 and find a 95% confidence interval for its P-value.

(3) Table 4.4 shows the frequency of one letter words per 1000 words in eight essays by A. Hamilton and seven by J. Madison.

Hamilton	24	21	23	24	33	28	28	37
Madison	20	27	19	30	11	17	27	

Table 4.4. Word frequencies: F. Mosteller and D. L. Wallace, *Inference and Disputed Authorship: The Federalist*. 1964, p. 248.

Our hypotheses:

- H: These are samples from the same distribution.
- A: These are samples from different distributions.

Find the P-value with `permSim.py`. The data is stored in `essays.txt`.

(4) Apply `permCalc.py` to Table 4.4. Find the P-value.

(5) Write a program that selects a random subset from the fourth row of the smoking data in Table 4.1 with sum T. It should increment a counter whenever $T \leq 10$ or $T \geq 64$, and repeat the selection task NUM_TRIALS times. At the end, it should estimate the P-value by calculating count/NUM_TRIALS/2 and compare the result with the exact value.

4.5 The Two Sample Problem of Wilcoxon

The Wilcoxon rank-sum test (also known as the Wilcoxon-Mann-Whitney test) determines whether two samples are drawn from the same distribution. It's best employed when we can determine the size of any difference between pairs of observations, as well as the direction of the difference.

For those with some statistics background, this may sound like Student's T-test, but that's used when both groups are normally distributed; Wilcoxon makes no assumptions about the underlying distribution.

The two names for the test are historical. Frank Wilcoxon proposed it in 1945, but a thorough analysis, including the recurrence relation we use below,

appeared in a 1947 article by Henry Mann and Donald Ransom Whitney: 'On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other', *The Annals of Mathematical Statistics*, Vol. 18, No.1 (Mar., 1947), pp. 50-60.

4.5.1 Data Analysis. We're going to examine the relationship between the heights and ages of death of the 35 US presidents who died of natural causes, based on data obtained from `https://potus.com/presidential-facts/`. The details are stored in CSV form in `presidents.csv`:

```
No,Name,Age,Height
1,George Washington,67,188
2,John Adams,90,170
3,Thomas Jefferson,83,189
4,James Madison,85,163
  : # more lines, not shown
38,Gerald R. Ford,93,183
40,Ronald Reagan,93,185
41,George H.W. Bush,94,188
```

The data is loaded and analyzed by `readPresidents.py` which generates a stem-leaf plot of the ages, a division of the ages into 'short' and 'tall' presidents, where 'short' is deemed to be < 173cm, a binary rank string, and two box-plots shown in Fig. 4.3.

The printed output:

```
> python readPresidents.py
Loading: presidents.csv
Keys: ['No', 'Name', 'Age', 'Height']
No. of records: 35

Presidential Ages
stem   leaf
   5 | 377
   6 | 003344567778
   7 | 0112347889
   8 | 01358
   9 | 00334

Short Ages: [67, 79, 80, 85, 90]
Tall Ages: [53, 57, 57, 60, 60, 63, 63, 64, 64, 65,
66, 67, 67, 68, 70, 71, 71, 72, 73, 74, 77, 78, 78,
81, 83, 88, 90, 93, 93, 94]

Ages as binary:
35 : 00000000000010000000000011001010000
Short Positions (from 1):
```

```
13 25 26 29 31
Rank sum: 124
```

The box-plots suggest that the 'short' and 'tall' groups might be two different populations, which we're going to investigate.

The binary rank string:

```
00000000000010000000000011001010000
```

is obtained by sorting the ages into ascending order, and representing a 'tall' president by 0 and a 'short' one by 1. It also indicates quite strongly that short presidents live longer. So we'll consider two hypotheses:

- H : Height has nothing to do with the age of death. The pattern is just a random fluctuation.
- A : Short presidents on average do live longer.

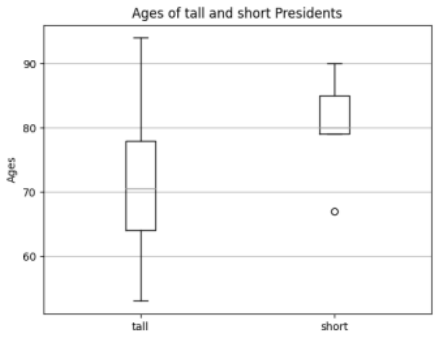


Figure 4.3. Box plot of presidential ages

readPresidents.py finishes by printing the positions of the 'short' presidents in the binary string (counting from the left, and starting at 1), and also the *rank sum* RS, the sum of those positions (124).

In the next two subsections we'll determine the P-value $= P(RS \geq 124|H)$. Our first approach approximates an answer through simulation, then we'll obtain an exact value using the Mann-Whitney recurrence relation.

4.5.2 An Approximate P-value. simupres.py chooses m of the ranks 1 to n at random and sums them. This is repeated NUM_TRIALS times, and a counter is incremented whenever the event RS $\geq$ max occurs.

When supplied with $n = 35$, $m = 5$, max $= 124$:

```
> python simupres.py
n m max=? 35 5 124
count == 1804; P == 0.05637500
```

We could also find a confidence interval for the P-value as we did in Sec. 4.3.

4.5.3 Analyzing the Binary String. Before we tackle the large binary string of presidential ages, let's look at the analysis steps using something smaller. 01001 is a 5-bit string with 1's in the second and fifth positions, so its rank sum RS $= 2 + 5 = 7$. We'll determine the exact P-value by finding the number of binary strings with 3 zeros and 2 ones with an RS ≥ 7.

1	2	3	4	5	RS
0	1	0	0	1	7
0	0	1	0	1	8
0	0	0	1	1	9
0	0	1	1	0	7

Table 4.5. All the 5-bit strings with RS ≥ 7

As shown in Table 4.5; there are 4 strings that qualify.

The total number of 5-bit strings with 3 zeros and 2 ones is $\binom{5}{2} = 10$, so the exact P-value $= 4/10 = 0.4$.

This manual approach isn't feasible for larger binary strings, such as the presidential ages, which is why Mann-Whitney developed the recurrence relation $w(u, m, n)$ to count the number of strings with a RS $\geq$ some value.

Before we can use $w(u, m, n)$, we need to determine initial values for its variables. Two of them are simple: m is the number of 1's in the string and n is the number of 0's. u is more complex, based on the *z sequence* for the string.

01001 can be encoded by a sequence $\{z_1, z_2\}$ of integers which count the number of zeros to the right of the first and second 1's in the string, counting to its end. So for 01001, $z_1 = 2$ and $z_2 = 0$. As another example, 11000 would be represented by $z_1 = 3$ and $z_2 = 3$. Note, that the length of the sequence equals the number of 1's in the string.

z_1 and z_2 can be converted into string positions for the 1's with the formula:

$$p_i = i + (n - z_i)$$

n is the number of 0's in the string ($n = 3$ here). So the position of the first 1 in 01001 is at $p_1 = 1 + (3 - z_i) = 1 + (3 - 2) = 2$ and the second 1 is at $p_2 = 2 + (3 - z_2) = 2 + (3 - 0) = 5$. Calculating the string's rank sum is now easy:

$$RS = \sum_{i=1}^{m} p_i = \sum_{i=1}^{m} (i - (n - z_i)) = \frac{m(m+1)}{2} + mn - \sum_{i=1}^{n} z_i$$

m is the number of 1's in the binary string ($m = 2$).

Recall that for 01001 we want to find the number of binary words with 3 zeros and 2 ones with rank sum RS ≥ 7, so substitute these into the RS equation

$$\frac{2 \cdot 3}{2} + 2 \cdot 3 - \sum_{i=1}^{n} z_i \geq 7$$

$$9 - \sum_{i=1}^{n} z_i \geq 7$$

$$\sum_{i=1}^{n} z_i \leq 2$$

This gives us an upper bound for $\sum_{i=1}^{n} z_i$. Mann-Whitney calls this u, the number of times a 1 precedes a 0 in the binary string. A look back at Table 4.5 shows that $u \leq 2$ for strings of this type.

$u = 2$ becomes the first argument of the recurrence relation $w(u, m, n)$ which calculates the number of strings containing n 0's and m 1's where a 1 precedes a 0 u times.

In this example, $m = 2$ and $n = 3$. However, n cannot exceed u, so can be reduced to $n = 2$. This reduction is welcome since it shortens the number of rewrites in the recursion. So our aim is to evaluate $w(2, 2, 2)$.

The three base cases for $w()$ are:

$$w(u, m, n) = 0 \text{ for } u < 0, \quad \text{and} \quad w(0, m, n) = w(u, m, 0) = 1,$$

In the recursive case, a string counted by $w(u, m, n)$ must deal with a sequence of 0's and 1's. When a 0 is met, the n count is decremented. When a 1 is met, the m count is decremented, and the remaining number of 0's is deducted from u. In other words:

$$w(u, m, n) = w(u, m, n - 1) + w(u - n, m - 1, n);$$

This will be true even for $m = 1$ if we set $w(u, 0, n) = 1$ for non-negative u and n, which takes the number of base cases to 4.

The base cases deal with u, m, or $n = 0$, which is equivalent to a count of 1. However, if u becomes negative it only signals termination of the search, so the base case returns 0.

Fig. 4.4 shows the evaluation of w(2,2,2) using the above definitions. It returns 4, the number of 5-bit strings where RS ≥ 7, and matches our manual calculation back in Table 4.5.

Now we'll apply the same analysis to the president ages string. Recall the details generated by `readPresidents.py`:

```
Ages as binary:
35 : 00000000000010000000000011001010000
Short Positions (from 1):
```

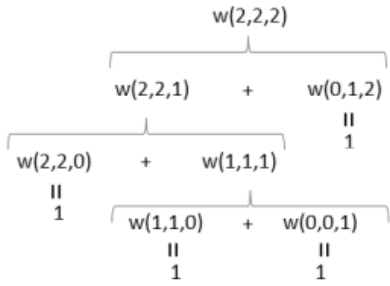


Figure 4.4. Evaluation of w(2,2,2)

```
13 25 26 29 31
Sum: 124
```

Once we've worked out the arguments for $w(u, m, n)$, we'll (happily) pass them to a program to do the counting.

The rank sum RS for the string is $13 + 25 + 26 + 29 + 31 = 124$, so we want to calculate the number of 35-bit strings where RS ≥ 124.

We start with the RS equation:

$$RS = \frac{m(m+1)}{2} + mn - \sum_{i=1}^{n} z_i$$

m is the number of 1's in the binary string ($m = 5$), and n the number of 0's ($n = 30$). Substitute these values into the equation:

$$\frac{5 \cdot 6}{2} + 5 \cdot 30 - \sum_{i=1}^{30} z_i \geq 124$$

$$165 - \sum_{i=1}^{30} z_i \geq 124$$

$$\sum_{i=1}^{30} z_i \leq 41$$

In $w(u, m, n)$, $u = 41$. m is the number of 1's in the string, so $m = 5$. n is the number of 0's, so $n = 30$, which is already less than u so doesn't change.

The straightforward translation of the $w()$ relations yields Listing 4.6 (wilcorec.py).

```
def w(u,m,n):
  if u < 0:
    return 0
  elif (u == 0) or (m == 0) or (n == 0):
    return 1
```

```
  else:
    return w(u, m, n-1) + w(u-n, m-1, n)
```

Listing 4.6. A recursive implementation of w()

Its calculation:

```
> python wilcorec.py
u m n=? 41 5 30
count = 18909
```

The total number of binary strings with 30 zeros and 5 ones is $\binom{35}{5}$, which is:

```
>>> import math
>>> math.comb(35,5)
324632
```

So the exact P-value = 18909/324632 = 0.05824749. By simulation we got P = 0.05637500.

4.5.4 Faster Implementations. Listing 4.7 (wiliter1.py) is a translation of $w()$ to use iteration.

```
u,m,n = map(int, input("u m n=? ").split())
w = [[[0 for y in range(n+1)]
       for x in range(m+1)] for z in range(u+1)]
for z in range(u+1):
  for x in range(m+1): w[z][x][0] = 1
for z in range(u+1):
  for y in range(n+1): w[z][0][y] = 1
for x in range(m+1):
  for y in range(n+1): w[0][x][y] = 1
for x in range(1,m+1):
  for y in range(1,n+1):
    for z in range(1,u+1):
      if z < y:
        w[z][x][y] = w[z][x][y-1]
      else:
        w[z][x][y] = w[z][x][y-1] + w[z-y][x-1][y]
print("w =", w[u][m][n])
```

Listing 4.7. An iterative w() using lots of memory

For larger problems it's considerably faster than the recursive function, with time complexity proportional to $u * m * n$.

Unfortunately it requires $(u + 1) * (m + 1) * (n + 1)$ cells for the 3D list $w[u][m][n]$, but this storage can be reduced considerably.

The algorithm computes $w(z, x, y)$ for $y = 0, 1, 2, \ldots$. At the start, the layer $y = 0$ is set up by assigning the value $w = 1$ to each lattice point (z, x) with $0 \le z \le u$, $0 \le x \le m$ (this follows the axis labeling in Fig. 4.5).

We can now compute the w-values at the lattice points for layers $y = 1, 2, ..., n$. Each computed value is stored in the current y layer, so the key insight is that we only need a single y layer. Thus, the recursion

$$w(z, x, y) = w(z, x, y-1) + w(z-y, z-1, y)$$

reduces to

$$w(z, x) = w(z, x) + w(z-y, x-1) \quad \text{for } z \geq y;$$

Note that the y index needs to be retained for the $z - y$ calculation, but there's no need to generate y layers. These changes are represented by Fig. 4.6.

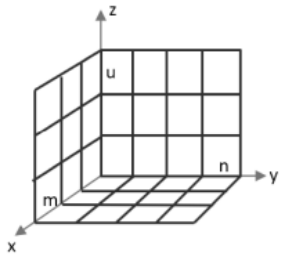


Figure 4.5. The 3D space used by w(u,m,n)

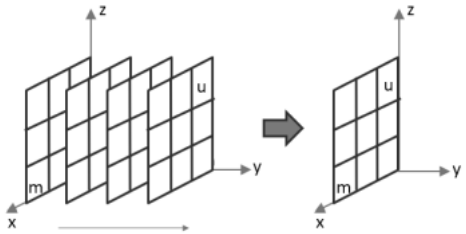


Figure 4.6. The 2D space used by w(u,m)

Listing 4.8 (wiliter2.py) implements the new algorithm.

```
u,m,n = map(int, input("u m n=? ").split())
w = [[0 for x in range(m+1)] for z in range(u+1)]
for z in range(u+1):
  for x in range(m+1): w[z][x] = 1
for y in range(1,n+1):
  for x in range(1,m+1):
    for z in range(y,u+1):
      w [z][x] = w[z][x] + w[z-y][x-1]
print("w =", w[u][m])
```

Listing 4.8. An iterative w() using less memory

4.6 The General Two Sample Test

In the presidents example we lost some information by replacing ages by ranks, so let's repeat the analysis using ages.

Suppose we take a random 5-subset of the 35 ages. What is the probability that the sum s of these ages is $\geq 67 + 79 + 80 + 85 + 90 = 401$? Let $q(n, k, s)$ be

the number of k-subsets of an n-set $\{d_1, d_2, ..., d_n\}$ of non-negative integers, whose sum is $\leq s$. Then

$$\begin{array}{rcll} q(n,0,s) & = & 1 & \text{for } s \geq 0, \\ q(n,k,s) & = & 0 & \text{for } n < k \text{ or } s < 0, \end{array} \tag{4.1}$$

and

$$q(n,k,s) = q(n-1,k,s) + q(n-1,k-1,s-d_n). \tag{4.2}$$

In words, Equ. (4.2) states that there are $q(n-1,k,s)$ subsets without d_n and $q(n-1,k-1,s-d_n)$ subsets with d_n.

To use this recursive definition, we'll need to 'reflect' the ages by subtraction (x' = maximum age $- x$) to make the ages of the short presidents small. Instead of $s \geq 67+79+80+85+90 = 401$ we'll have $s \leq 4+9+14+15+27 = 69$ (the maximum age is 94).

Equs. (4.1) and (4.2) become the $q()$ function in Listing 4.9 (TwoSample.py).

```
def q(n, k, s):
  if (k == 0) and (s >= 0):
    return 1
  elif (n < k) or (s < 0):
    return 0
  else:
    return q(n-1, k, s) + q(n-1, k-1, s-d[n-1])
```

Listing 4.9. Implementation of $q()$

The output:

```
> python TwoSample.py
n=35, k=5, s=69
P = 7.48%
```

We note that rank testing of the same data gave us a P-value which was too good (around 5.82%). In general that approach tends to overestimate tiny differences since every value gets the same weight.

Exercises for Sections 4.5 to 4.6

(1) Prove that $w(u,m,n) = w(u,n,m)$.

(2) Write iterative programs for the Two Sample Test imitating Listings 4.7 and 4.8.

(3) Find the P-value of the hypothesis in Ex. 3 at the end of Sec. 4.4 by means of Listings 4.8 and TwoSample.py.

(4) Test the presidents data with permCalc.py. You should step through all the 5-subsets of the 35 samples and count those with sums ≥ 401. Compare with the output of TwoSample.py.

(5) Apply the Two Sample test to the marijuana example. It should give the same result as permSim.py, but faster. However, you should first turn the data into non-negative integers by adding 17 to each number. The input will still be $n = 18$ and $k = 9$, but $s = 9 * 17 - 46 = 107$. To gain even more speed you should use the programs suggested in Ex. 2 above.

4.7 Kendall's Rank Correlation

Kendall's rank correlation (also known as Kendall's τ (Tau)) is used to measure the ordinal association between two sets of data. The correlation will be high when the values in the two sets have similar rank positions, and low when their ranks are dissimilar.

We'll explain how this statistic is applied by considering cancer deaths associated with the Hanford site, a decommissioned nuclear production complex operated by the US government in Hanford, Washington. Although its reactors were shut down between 1964 and 1971, engineers sent to clean up the installation discovered 54 million gallons of highly radioactive sludge. Until 1965 this waste had been stored in open pits, which leaked into the Columbia river that flows through Oregon into the Pacific. Cleanup activity are still ongoing to this day, with over 10,000 workers employed on various tasks.

Table 4.6 shows how an exposure index X compares to cancer mortality Y (per 100000 man years for 1959-1964) for nine Oregon districts. Note that the data is sorted by the exposure index.

X	1.25	1.62	2.49	2.57	3.41	3.83	6.41	8.34	11.64
Y	113.5	137.5	147.5	130.1	129.9	162.3	177.9	210.3	207.5

Table 4.6. 'Exposure and Mortality', *Journal of Environmental Health*, v. 27, 1965, 883-897.

If we replace each cancer mortality value by its rank, we get the permutation

$$p = 145326798$$

The first tabulated Y value has rank 1, the second has rank 4, the third rank 5, etc. Is this a 'random permutation'? Obviously, 123456789 and 987654321 are not random, but what about 145326798? Kendall's Tau can be used to answer this question.

Consider the $\binom{9}{2} = 36$ pairs of ranks that can be picked from p. A pair (i, j) with $i < j$ is called *concordant*, while a pair with $i > j$ is *discordant*. 123456789 has no discordants, while 987654321 has 36, but on average, we'd expect 18 concordant and 18 discordant pairs in a random permutation. We'll use the number of discordants as a way to measure 'extremeness'.

$p = 145326798$ has 30 concordants and 6 discordants, as calculated in Table 4.7.

Exposure Index	Cancer mortality rank	Concordants	Discordants
1	1	8	0
2	4	5	2
3	5	4	2
4	3	4	1
5	2	4	0
6	6	3	0
7	7	2	0
8	9	0	1
9	8	0	0
	Sum	30	6

Table 4.7. Calculating concordants and discordants for cance mortality

To count the number of concordant pairs, start with the first row of the 'Cancer mortality' column and count how many ranks below it are larger. Then move to the next row and repeat the process. To count the number of discordant pairs, start again at the top of the 'Cancer mortality' column and count how many ranks below it are *smaller*. Repeat this for every row. The 'Concordants' and 'Discordants' totals are 30 and 6 respectively, which together equals the total number of ways of selecting two pairs; in this case $\binom{9}{2} = 36$.

Carrying out a similar task for $p' = 897623541$ results in 6 concordants and 30 discordants, which is to be expected since

$$P(\text{discords}(p) \leq 6|H) = P(\text{discords}(p) \geq 30|H),$$

where

- H: It is just a random fluctuation.
- A: p has too few (or too many) discordants.

We calculate this probability in cancer.py. It generates NUM_TRIALS random 9-permutations, totals their discordants, and tests if discords(p) ≤ 6 or ≥ 30. For each occurrence of this event a counter is incremented, so that by the end the counter will record the number of 9-permutations with too many or too few discordants. This is converted to a probability by division by 9!. It's also divided by two since we're examining both ends of the distribution so the probability range is doubled.

Typical output when $n = 9$, min $= 6$, max$= 30$:

```
> python cancer.py
n min max=? 9 6 30
count == 396; P == 0.00619
```

This is saying that there's a high significance that assumption A is true.

4.7.1 Defining Kendall using Recursion. Let $p(n,k)$ be the number of n-permutations with at most k discordants. Our aim is to derive a recursive formula for $p(n,k)$ that can be readily translated into a recursive function.

Suppose we have an $(n-1)$-permutation $X_1X_2...X_{n-1}$ of {1, 2, ..., n-1}, and we want to insert the element n to make it an n-permutation. There are n possible insertion points, as indicated by the boxes in:

$$\boxed{n-1}\,X_i\,\boxed{n-2}\,X_2\,...\boxed{i}\,X_{n-i}...\boxed{2}\,X_{n-2}\,\boxed{1}\,X_{n-1}\,\boxed{0}.$$

If we insert element n into location i, then it will contributes i discordants. Our aim is to obtain $p(n,k)$, so the $p()$ value for the old $(n-1)$-permutation must be $p(n-1,k-i)$. There are n alternatives like this that can produce $p(n,k)$, leading to the formula:

$$p(n,k) = p(n-1,k) + p(n-1,k-1) + ... + p(n-1,k-n+1). \tag{4.3}$$

If we substitute $k-1$ in place of k in Equ. (4.3), we get

$$p(n,k-1) = p(n-1,k-1) + p(n-1,k-2) + ... + p(n-1,k-n). \tag{4.4}$$

The right-hand side of Equ. (4.4) is almost the same as the right-hand side of Equ. (4.3), which means that (4.3) can be simplified by substituting in the left-hand side of (4.4), giving:

$$p(n,k) = p(n-1,k) + p(n,k-1) - p(n-1,k-n). \tag{4.5}$$

Recalling that $p(n,k)$ is the number of n-permutations with at most k discordants, we specify three base cases:

$$\begin{aligned} p(n,0) &= p(1,k) = 1 \text{ for } n \geq 1, k \geq 0, \\ p(n,j) &= 0 \text{ for } j < 0 \end{aligned}$$

The recursive case in Equ. (4.5) can also be simplified when $k < n$ since the third term will have a negative argument and so contribute 0. Therefore:

$$p(n,k) = p(n-1,k) + p(n,k-1) \text{ for } k < n$$

These three base cases, and two recursive rules, become Listing 4.10 from kendallr.py.

```
def numDiscs(n,k):
  if (k == 0) or (n == 1):
    return 1
  elif k < n:
    return numDiscs(n-1, k) + numDiscs(n, k-1)
  else:
```

```
    return numDiscs(n-1,k) + numDiscs(n,k-1) - numDiscs(n-1, k-n)
```

Listing 4.10. Recursive Kendall's Tau

For $n = 9$, $k = 6$, we get:

```
> python kendallr.py
n k=? 9 6
no. discordants = 2298
p = 0.00633
```

The iterative version, kendallit.py, is faster for larger problems.

```
def numDiscs(n,k):
  nds = [[0 for j in range(k+1)] for i in range(n+1)]
  for i in range(1,n+1):
    nds[i][0] = 1
  for j in range(k+1):
    nds[1][j] = 1
  for i in range(2,n+1):
    for j in range(1,k+1):
      if j < i:
        nds[i][j] = nds[i][j-1] + nds[i-1][j]
      else:
        nds[i][j] = nds[i][j-1] + nds[i-1][j] - nds[i-1][j-i]
  return nds[n][k]
```

Listing 4.11. Iterative Kendall's Tau

4.7.2 The Permutation Test and Correlation. Let us look again at Table 4.6, but with a test statistic of our own invention, the *dot product* of two sequences X and Y:

$$T = X \cdot Y = \sum X[i]\, Y[i].$$

The data in X is always sorted into ascending order, so T will be a maximum if the elements of Y are also in ascending order, and decreases as the number of discordants in Y increases.

cancer1.py calculates the dot correlation between the X and Y sequences in Table 4.6, and then repeatedly shuffles the Y sequence and retests it against X.

A count is kept of each instance when a larger correlation is calculated, which is used to estimate the probability that a random correlation could be higher than the one obtained between X and Y.

Typical output:

```
> python cancer1.py
X: [1.25, 1.62, 2.49, 2.57, 3.41, 3.83, 6.41, 8.34, 11.64]
Y: [113.5, 137.5, 147.5, 130.1, 129.9, 162.3, 177.9, 210.3, 207.5]
dot corr = 7440.3660
count = 15; p = 0.00047
```

As with cancer.py the result is highly significant, indicating that the correlation of $Y$with X is very unusual.

4.7.3 The Sample Pearson Correlation Coefficient. One way to define the Pearson correlation coefficient r_{xy} over paired data $(x_1, y_1), ..., (x_n, y_n)$ is

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

We're assuming that all the summations are between 1 and n. Listing 4.12 (corrCoef.py) implements this equation.

```
def dot(xs, ys):
  return sum([x*y for x, y in zip(xs, ys, strict=True)])

def corrCoef(xs, ys):
  # Assume len(xs) == len(ys)
  n = len(xs)
  xSum = sum(xs)
  ySum = sum(ys)
  xSum2 = sum([x*x for x in xs])
  ySum2 = sum([y*y for y in ys])
  xys = dot(xs, ys)
  numer = n*xys - xSum*ySum
  denom = math.sqrt((n*xSum2 - xSum*xSum) * (n*ySum2 - ySum*ySum))
  if denom == 0:
    return 0
  return numer/denom
```

Listing 4.12. The Pearson correlation coefficient

Listing 4.12 is a near duplication of the correlation() function in Python's statistics module, but that version is more versatile. For instance, if its 'method' argument is set to 'ranked', it will switch to calculating Spearman's rank correlation coefficient.

corrCoef() is utilized in cancer2.py which is identical to cancer1.py except that the two calls to dot() are replaced by calls to corrCoef(). Typical output:

```
> python cancer2.py
X: [1.25, 1.62, 2.49, 2.57, 3.41, 3.83, 6.41, 8.34, 11.64]
Y: [113.5, 137.5, 147.5, 130.1, 129.9, 162.3, 177.9, 210.3, 207.5]
pearson corr = 0.9259
count = 10; p = 0.00031
```

The significance level 0.00031 is almost the same as the dot correlation approach above. The difference is probably due to Pearson's underlying assumption that the data is normally distributed (which is implicit in its use of means and standard deviations in its summations), whereas dot correlation makes no such assumptions.

Exercises

(1) draft.csv holds US draft lottery data for 1970, which determined the order of calls to military service in the Vietnam war that year for men born between January 1, 1944 and December 31, 1950 (`https://en.wikipedia.org/wiki/Draft_lottery_(1969)`). readDraft.py can be used to load it as a list of Python dictionaries.

For the draft, 366 days of the year (including February 29) were written on slips of paper, placed in individual plastic capsules, mixed together in a shoe box(!), then emptied into a glass jar. Capsules were drawn from the jar one at a time and opened. The first number drawn was 258 (for September 14), so all registrants with that birthday were assigned rank 1. The next ranks were assigned to April 24, December 30, February 14, October 18, and so on. The last number drawn was for June 8. In draft.csv, we've replaced the birthday numbers by their selection rank. So September 14 is 001, April 24 is 002, and so on.

All of the men who shared a birthday were called to serve, and altogether 195 birthdays were utilized; the last being September 24. Thus, men with low rankings were almost certainly drafted, while those with high ranks were spared. Our aim is to show that the distribution of the ranks in draft.csv is not random.

For each month, we found the median of the ranks assigned to its days, resulting in Table 4.8.

month	1	2	3	4	5	6	7	8	9	10	11	12
median of ranks	211	210	256	225	226	207.50	188	145	168	201	131.5	100

Table 4.8. Draft ranks per month

a) Assign a rank to each median in Table 4.8, producing a permutation of the numbers 1 to 12.

b) Determine the number of concordant pairs, C. You should get $C = 11$, but the expected value is 33 (why 33?).

c) Test the hypothesis H that the rankings in draft.csv are random. Determine the probability $P(C \leq 11|H)$ and draw conclusions.

It seems very likely that the plastic capsules holding the birth dates weren't shuffled sufficiently in the shoe box. The last ones put in tended to be the first ones pulled out.

(2) The archived website at `https://web.archive.org/web/20050318170452/http://cnx.rice.edu/content/m10142/latest/` by David Lane also examines the fairness of the 1970 US draft lottery. He starts with a list of the selected birthday numbers:

```
258 115 365 45 292 250 300 251 ...  52 126 57 160
```

This data can be found in draftBithdays.txt.

He divided the 366 birth dates into three groups of 122 days each: the first third goes from January 1 to May 1, the second from May 2 to August 31, and the last from September 1 to December 31. He plots these groups as three box plots, noting that if there was no relationship between birth date and draft number, then the three boxes would line up horizontally; they don't.

Write a program to load the data from `draftBirthdays.txt` (using the functions in reader.py, split the data as Lane suggests, and use matplotlib to produce three box plots. A similar charting approach was used by readPresidents.py in Sec. 4.5.

(3) Table 4.9 gives the cholesterol levels of males in two age groups. Test the fol-

X (20-30 years)	135	222	251	260	269	235	386	252	173	156	352
Y (40-50 years)	294	311	286	264	277	336	208	346	173	254	346

Table 4.9. Cholesterol levels. Source: Dixon and Massey, *Introduction to Statistical Analysis*.

lowing hypotheses:

- H : The cholesterol levels of both groups is the same.
- A : The older group has higher cholesterol levels.

a) Find the P-value using TwoSample.py.

b) Find the P-value using permSim.py.

4.8 The Binomial Distribution

Fig. 4.7 shows a spinner where p and q $(1-p)$ are the probabilities of the outcomes 1 (success) and 0 (failure).

With this distribution, any binary word, such as 100110...01 with x ones and $n-x$ zeros, will have probability $p^x q^{n-x}$ of occurring. Since there are altogether $\binom{n}{x}$ such words, the probability of exactly x successes in n spins (trials) is

$$b(x) = \binom{n}{x} p^x q^{n-x}$$

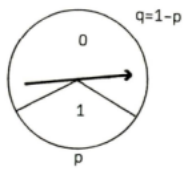


Figure 4.7. A 1 with prob p

If we make explicit that the values of n and p can change, then the equation becomes

$$b(n, p, x) = \binom{n}{x} p^x q^{n-x}, \quad x = 0, 1, 2, ..., n. \tag{4.6}$$

This, with the help of math.comb(), becomes Listing 4.13 (binomial.py).

```
def binomial(n, p, x):
  return math.comb(n, x) * p**x * (1-p)**(n-x)
```

Listing 4.13. Binomial using float

The drawback of this approach is the danger of float overflow or underflow (see Appendix E). For example:

```
>>> from binomial import *
>>> binomial(100, 0.5, 0)
7.888609052210118e-31
>>> binomial(1000, 0.5, 0)
9.332636185032189e-302
>>> binomial(1080, 0.5, 0)
0.0
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

As sys.float_info states, floats can only deal with numbers between around 2×10^{-308} and 2×10^{308}.

One solution is to rewrite binomial() to use Python's decimal type. Let's switch back to the $b(x)$ form, and the transformation is more apparent if we write $b(x)$ and $b(x-1)$ in terms of factorials:

$$\begin{aligned} b(x) &= \frac{n!}{(n-x)!\,x!} p^x q^{n-x} \\ b(x-1) &= \frac{n!}{(n-(x-1))\,(x-1)!} p^{x-1} q^{n-(x-1)} \end{aligned}$$

Defining $b(x)$ in terms of $b(x-1)$:

$$b(0) = q^n, \quad b(x) = b(x-1) * r * \frac{n-x+1}{x}, \quad r = \frac{p}{q}, \quad x = 1, 2, ..., n. \tag{4.7}$$

Equ. (4.7)'s recursion can be readily translated into a loop, and floats are converted to decimals, to produce Listing 4.14 (binomial.py).

```
def binomialIterD(n, p, x):
  dx = D(x); dp = D(p)
  dq = D(1)-dp
```

```
  bin = dq**n
  for i in range(1, x):
    bin *= dp/dq * (n - dx + 1)/dx
  return bin
```

Listing 4.14. Binomial using decimals

Underflow and overflow are avoided:

```
>>> binomialIterD(100, 0.5, 0)
Decimal('7.888609052210118054117285653E-31')
>>> binomialIterD(1000, 0.5, 0)
Decimal('9.33263618503218878990089544 7E-302')
>>> binomialIterD(1080, 0.5, 0)
Decimal('7.719775716269477252758887389E-326')
```

4.8.0.1 A Confidence Interval for p. A random process is controlled by the spinner in Fig. 4.8.

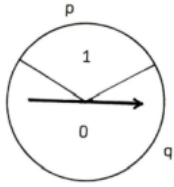


Figure 4.8. Obtain a 1 with prob p

If we carry out n spins and get X successes, then $\hat{p} = X/n$ is a *point* estimate for p. However, it's more useful usually to state a level of confidence that p lies within some range of probabilities.

Suppose we get $x = 80$ successes in $n = 200$ trials, then $80/200 = 0.4$ is the point estimate for p. Of course we could get fewer than 80 successes even if p was larger than 0.4, but as p grows this becomes less likely. Let's find an upper limit $\hat{p}_2$ for p where $P(X \leq 80) = 2.5\%$ exactly. The equation we must solve is

$$B(p) = P(X \leq 80) = \sum_{x=0}^{80} \binom{200}{x} p^x (1-p) p^{200-x} = 0.025.$$

We do this by testing different p values with Listing 4.15 (binomialSum.py).

```
import math
from binomial import *

n,p,c,d = map(float, input("n p c
    d=? ").split())
n = int(n); c = int(c); d = int(d)
    ; sum = 0
for x in range(c, d+1):
  sum += binomial(n, p, x)
print(f"{sum:.8f}")
```

Listing 4.15. Summing binomials

For example:

```
> python binomialSum.py
n p c d=? 200 0.45 0 80
0.08807268

> python binomialSum.py
n p c d=? 200 0.47 0 80
0.02751749
```

The inputs $n = 200$, $c = 0$, $d = 80$ and different p values eventually lead us to $\hat{p}_2 = 0.47147$, as shown in Table 4.10.

p	0.45	0.47	0.471	0.472	0.4714	0.47147
B(p)	0.0881	0.0275	0.0258	0.0241	0.0251	0.0250

Table 4.10. Outputs from Listing 4.15

So if $p > 0.47147$ then the probability of getting only 80 successes in 200 trials is 2.5% or less. We can also phrase this as a 97.5% confidence that $p \leq 0.47147$. So if we repeatedly carry out experiments with this confidence level, we shall be successful, in the long run, at least 97.5% of the time.

It may seem that we could have expressed the same thing more simply by saying that the probability that $p \leq 0.47147$ is at least 97.5%. However, strictly speaking, this type of statement is appropriate only when p is a random variable which can take various values. But our p has a single well defined value, we just don't know what it is. Our calculated probability doesn't refer to the value of p, but to the correctness of the statement we are making about it. That's why we use the word "confidence".

If p was much smaller than 0.4, it's unlikely that we would have got 80 or more successes. We can quantify that with the equation:

$$B(p) = P(X > 80) = \sum_{x=80}^{200} \binom{200}{x} p^x (1-p) p^{200-x} = 0.025.$$

A suitable lower limit for p can be found using Listing 4.15 but with the summation range $c = 80$ and $d = 200$. We obtain:

```
> python binomialSum.py
n p c d=? 200 0.33155 80 200
0.02500667
```

Let $\hat{p}_1 \approx 0.33155$, which means that if $p < \hat{p}_1$ then the probability of obtaining 80 or more successes is $< 2.5\%$. In other words, $\hat{p}_1 < p$ is also a 97.5% interval for p. Therefore the intersection

$$0.33155 \leq p \leq 0.47147$$

of the two intervals is a 95% confidence interval.

4.8.1 Plotting the Binomial Distribution. plotBinomial.py generates a bar plot of binomial(n, p, x) values for $0 \leq x \leq n$, which illustrates how the distribution begins to approximate a Normal distribution when n is large enough and p is close to 0.5. The result for $n = 50$ and $p = 0.5$ is shown in Fig. 4.9.

It's fairly clear that the mean of the Normal distribution is 25 (i.e np) and one standard deviation is 5 (i.e. npq).

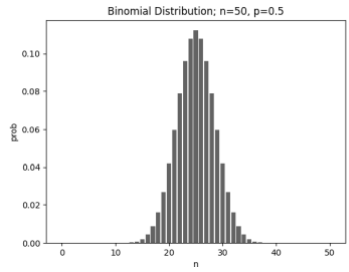
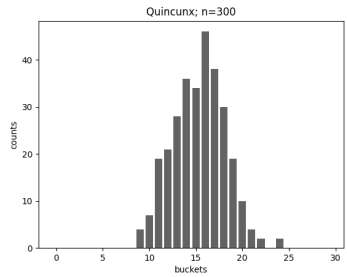


Figure 4.9. Plot of a Binomial distribution



(a) A quincunx



(b) Plot of a quincunx

Figure 4.10. Binomials

Another way of linking the Binomial and Normal distributions is by implementing a Galton board (see Fig. 4.10a), also known as a *quincunx*. Francis Galton used it to demonstrate the central limit theorem: with a sufficiently large sample size the Binomial distribution approximates a Normal distribution.

quincunx.py implements the Galton board, not by calling our binomial() function but by tallying the results of multiple random yes/no choices. This approach hopefully emphasizes that the binomial is the discrete probability distribution of the number of successes in n independent experiments, each asking a yes/no question. When $n = 200$, the resulting plot is Fig. 4.10b.

Exercises

(1) For the example in the text with $X = 80$, $n = 200$ find a confidence interval for p of a) 98%, b) 99%.

(2) For $X = 50$ and $n = 100$ find a confidence interval for p of a) 95%, b) 98%.

(3) An unknown proportion p of a population are smokers, but in a random sample of 1000 people 600 admitted to the habit. Find a confidence interval for p of a) 95%, b) 98%.

(4) We want to estimate the proportion p of left-handers among students. In a random sample of $n = 100$ there were $X = 10$ left-handers. Find 95% and 98% confidence intervals for p.

(5) 100 coin tosses results in $X = 60$ successes. To see how likely this is if the coin is fair, find the probability of the event $|X - 50| \geq 10$ for such a coin.

(6) 900 coin tosses results in $X = 486$ successes. Test the likelihood of this occurring by computing the probability of getting $X \geq 486$ when tossing a fair coin.

(7) 600 rolls of a die results in $X = 120$ sixes. How likely is this if the die is fair? What is the probability of $|X - 100| \geq 20$?

(8) **The Taxi Problem**. City taxis are usually numbered from 1 to T. We want to find a 95% confidence interval (CI) for this unknown T by observing n taxi numbers, which we've ordered according to size: $X_1 \leq X_2 \leq ... \leq X_n = M$ = the biggest observed taxi number. This equation uses $\leq$ since the same taxi may show up repeatedly.

By definition, M $\leq$ T, so we take M as the lower end of the CI. We find the upper end by using $P(\text{all } X \leq M) = (M/T)^n$. We want to exclude T values where this probability is $< 5\%$ so we set the upper end to be $T = M\sqrt[n]{20}$. This comes from:

$$\left(\frac{M}{T}\right)^n > \frac{1}{20}$$
$$\left(\frac{T}{M}\right)^n > 20$$
$$T < M\sqrt[n]{20}$$

Combining both ends of the CI, we want $M \leq T \leq M\sqrt[n]{20}$ to be wrong only for 5% of taxi-cabbed cities.

Write a program which chooses n taxi numbers at random using some T, calculates the CI, and checks if it covers T. This should be repeated 1000 times. Does the CI cover T in about 95% of the cases? How does the CI depend on n?

It's easy to calculate an almost "best point estimate" $\hat{T}$ for T. We consider the n observed gaps between the taxis: $X_1 - 1, X_2 - X_1, X_3 - X_2, ..., X_n - X_{n-1}$. The lengths of all of these gaps, including the first and the last, have the same probability distribution, so we estimate the last unknown gap, $T - M$, as the

mean of the observed ones, which is $(M-1)/n$. So rearranging

$$\begin{aligned} T - M &= \tfrac{M-1}{n} \\ T &= M + \tfrac{M}{n} - \tfrac{1}{n} \\ \hat{T} &= M(1 + \tfrac{1}{n}) - \tfrac{1}{n} \end{aligned}$$

$\hat{T}$ is being used as the *gap estimate* for T. So your code should calculate $\hat{T}$ and T, and compare them.

4.9 Hypergeometric Distribution

The hypergeometric distribution describes the probability of k successes in n draws *without* replacement, from a finite population of size N that contains exactly K objects. The binomial distribution differs in that it describes the probability of k successes in n draws *with* replacement.

The hypergeometric is identical to the one-tailed version of Fisher's exact test, named after Ronald Fisher. It's *exact* in the sense that its P-value can be calculated exactly.

One large group of experiments where the hypergeometric distribution is appropriate is the investigation of new medical treatments (e.g. new drugs) where the experiment must be **controlled**, **randomized**, and **double blind**. This means:

(1) The effects of a treatment can be judged only by comparison. We need an *experimental group E* which receives the treatment, and a control group C, which does not receive the treatment (a *controlled experiment*).

(2) The only way to exclude extraneous factors is to assign subjects to groups E and C by a random device (*randomization*).

(3) A subject mustn't know if he/she belongs to group E or C. Hence each person in C receives a placebo, which has no effect, but can't be distinguished from the new treatment. This ensures that any effects are due to the medicine and not psychological (a *blind experiment*).

(4) The doctor who measures the effects of the treatment mustn't know if a subject belongs to group C or E. This ensures that the diagnosis isn't influenced by the doctor's subconscious behavior (a *double-blind experiment*).

Does vitamin C help to prevent the common cold? This question has been studied repeatedly with contradictory results. We'll use one of the more extensive and reliable data sets (Table 4.11) from the *Canadian Medical Association Journal*, Sept. 1972, 503-508 (Toronto Study). It was a controlled, randomized, and double-blind experiment.

	cold	no cold	sum
Group E	302	105	407
Group C	335	76	411
sum	637	181	818

Table 4.11. Vitamin C and colds

The 407 subjects in group E received a vitamin C pill daily, while the 411 subjects in the control group C were given a placebo, which tasted exactly like the vitamin.

To simplify the calculations, we'll concentrate on the smallest of the four numbers in Table 4.11. How many people should we expect to see in the '76' box, which denotes untreated people who stayed healthy?

A person picked at random will fall into the second row with probability $\hat{p_1} = 411/818$ and into the second column with probability $\hat{p_2} = 181/818$. The probability of one person falling into the '76' box is the intersection of those two sets, $\hat{p_1}\hat{p_2}$. For n people, we would expect $n\hat{p_1}\hat{p_2}$ people in that box, i.e.

$$n\hat{p_1}\hat{p_2} = 818 * \frac{411}{818} * \frac{181}{818} = \frac{411 * 181}{818} \approx 91.$$

It looks as though the difference between the expected number and the actual value (91 - 76 = 15) might be a random fluctuation, but appearances are sometimes deceiving. We must test the hypotheses:

- H : Vitamin C does not help.
- A: Vitamin C helps.

If H is true then the numbers indicate that $s = 181$ of 818 people remain uninfected for reasons unrelated to vitamin C. The results are just one possible outcome of a random experiment. Instead of recording 76 in the box, we can generalize to x, and modify the other three boxes to state their results in terms of x (see Fig. 4.11).

We've also included three variables, n, r, and s in Fig. 4.11 to denote the totals for certain groups: n = the total number of people in the experiment, r = the number of people in group C (the placebo group), and s = the total number of people who did not catch a cold. These variables allows us to rewrite the table to be completely general, resulting in Table 4.12.

To test our hypothesis, we observe that if vitamin C were effective, we'd expect fewer individuals in the untreated yet uninfected category (the x box) than we would on a random basis; the uninfected people would tend to be in the treated group (the $s - x$ box). Therefore our P-value will be the probability that

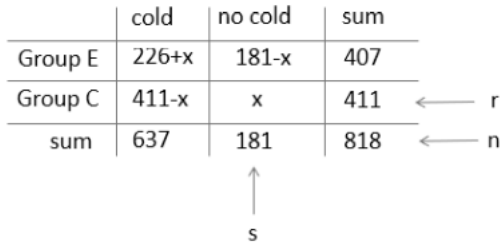


Figure 4.11. Vitamin C and colds with x untreated and healthy

	cold	no cold	sum
Group E	n-s-r+x	s-x	n-r
Group C	r-x	x	r
sum	n-s	s	n

Table 4.12. Vitamin C and colds expressed with variables

x is 76, as it was in the actual experiment, or smaller, by chance alone:

$$\text{P-value } = P(0 \le x \le 76|H) = \sum_{x=0}^{76} h(x).$$

The P-value is defined as the sum of the $h()$ function, which calculates the probability that there are x untreated people out of a total of s people who did not catch a cold. This breaks down into three counts related to people who did not catch a cold, highlighted in Fig. 4.12.

	cold	no cold	sum
Group E	n-s-r+x	s-x	n-r
Group C	r-x	x	r
sum	n-s	s	n

Figure 4.12. Counts involving people who did not catch a cold

The three counts are:

(1) $\binom{n}{s}$: the number of ways of select s (people with no colds) from n (the total number of people in the experiment).

(2) $\binom{r}{x}$: the number of ways of selecting x (the untreated, healthy people) from r (all the untreated people)

(3) $\binom{n-r}{s-x}$: the number of ways of selecting $s-x$ (the treated, healthy people) from $n-r$ (all the treated people)

h(x) combines these three counts to produce a probability for x given n, r, and s:

$$h(x) = \frac{\binom{r}{x}\binom{n-r}{s-x}}{\binom{n}{s}}, \quad h(0) = \frac{\binom{n-r}{s}}{\binom{n}{s}} = \frac{(n-r)\cdots(n-r-s+1)}{n\cdots(n-s+1)}$$

$h(0)$ follows immediately by setting $x = 0$ and considering the factorial results for the two choices in the numerator.

Listing 4.16 (leftTail.py), which employs math.comb(), follows directly from the definition for $h()$, and doesn't need to treat $h(0)$ as a special case.

```
def h(n, r, s, x):
  rs = math.comb(r, x)
  nrsx = math.comb(n-r, s-x)
  ns = math.comb(n, s)
  return (rs * nrsx)/ns

def hSum(n, r, s, x):
  return sum([h(n, r, s, i) for i in range(x+1)])
```

Listing 4.16. Summing $h()$ using floats

The inputs from the original problem $n = 818$, $s = 181$, $r = 411$, $x = 76$ yield

```
>>> from leftTail import *
>>> hSum(818, 411, 181, 76)
0.00742356691775225
```

In other words, the probability of getting 76 or less untreated, healthy people is $< 1\%$, which indicates that vitamin C definitely helps.

One drawback of this approach, which it shares with the binomial code from the last section, is it's use of Python floats, which means that underflow or overflow could be a problem, We'll address this in a similar way as before.

$h(x)$ can be defined recursively as

$$h(x) = h(x-1) * \frac{(r-x+1)(s-x+1)}{x(n-s-r+x)}$$

This is obtained by writing $h(x)$ and $h(x-1)$ in factorial form, and noting how $h(x-1)$ can be converted to $h(x)$ through various multiplications and divisions.

Although this formula is recursive, it can be easily converted to an iterative form, which we implement using Python's decimal type. The resulting functions, hD() and hSumD(), are included in leftTail.py . The inputs $n = 818$, $s = 181$, $r = 411$, and $x = 76$ yield

```
>>> hSumD(818, 411, 181, 76)
Decimal('0.007423566917752249768957316007')
```

Exercises

(1) Until near the end of the 19th century, mortality after surgery was extremely high. Even after surgeons finally switched from dirty to clean bandages, there wasn't much improvement. Finally the surgeon Joseph Lister started to use sterilized bandages. Table 4.13 shows the results of 75 amputations he performed, 35 without and 40 with sterilized bandages.

	steril.	unsteril.
survived	34	19
died	6	16

Table 4.13. Sterilized bandages and survival. From C.E.A. Winslow, *The Conquest of Epidemic Disease*. Princeton 1943, p.303.

For the hypotheses:

- H : Sterilization does not help
- A : Sterilization helps

find the P-value.

It's interesting to note that this statistical evidence did not immediately persuade other doctors to follow Lister's approach. The historical background is discussed in 'Statistics and the British controversy about the effects of Joseph Lister's system of antisepsis for surgery, 1867–1890', Ulrich Tröhler, *J R Soc Med.*, 2015 Jul; 108(7): 280–287.

(2) Modify Listing 4.16 to compute $P = h(x) + h(x+1) + ... + h(n)$, and solve the following problem. To estimate the number of trout in a pond, 1000 trout were caught, marked, and released. A few days later another 1000 trout were caught, and 100 turned out to be marked. Find a 95% confidence interval for the number N of trout in the pond.

Hint: proceed by trial-and-error to find confidence intervals for p in a similar way to in Sec. 4.8.

(3) During 1970-1974, exactly 194 patients with hip fractures were brought to a municipal hospital. Of the 73 patients delivered on Monday to Wednesday, none died. Of the 121 patients delivered Thursday to Sunday, ten died. Table 4.14 summarizes the data.

For the hypotheses:

- H : Death does not depend on the day of the week the patient was admitted

arrival	survived	died	sum
Mon-Wed.	73	0	73
Thu-Sun.	111	10	121
sum	184	10	194

Table 4.14. Hips and survival. Source: D. McNeil: 'Hip Fractures - Influence of Delay in Surgery on Mortality', *Wisconsin Medical Journal*, 74, Dec. 1975, 129-130.

- A : Patients arriving on weekends have a greater mortality rate (probably because their treatment was delayed)

find the P-value.

Why did we count Thursday as a weekend day? Surgery is treated as a last resort, so a patient arriving on a Thursday is observed for one to two days before a decision on progressing with surgery is made.

(4) **Quiet Don**. In 1965 M. Sholokhov received the Nobel Prize for literature for the novel *Quiet Don*, but rumors had been circulating in the USSR and abroad since 1928 that the novel wasn't his work. In an anonymous study by a Soviet critic the novel was attributed to the Cossak writer F. Kryukov who died of typhoid in 1920. The allegations were denounced as malicious slander in the official Soviet newspaper, *Pravda*.

In 1977 Geir Kjetsaa and others published research on the question. 1000 words were chosen at random from each of *Marking Time* (by Kryukov), *The Way and the Road* (by Sholokhov) and *Quiet Don* (?), and the number of lexemes counted. A lexeme is a single dictionary entry, so *write*, *writes*, *wrote*, *written*, and *writing* are treated as being the same. Table 4.15 shows the results.

	Words	Lexemes
Marking Time (Kryukov)	1000	589
The Way and the Road (Sholokhov)	1000	656
The Quiet Don (?)	1000	646

Table 4.15. Lexeme counts. Source: G. Kjetsaa, 'The Battle of the Quiet Don: Another Pilot Study', *Computers and the Humanities*, Vol. 11, pp. 341-346, 1977.

Since there's almost no difference between Sholokhov's writing and *Quiet Don*, Kryukov's writing was also tested against *Quiet Don*, resulting in Table 4.16.

For the hypotheses:

	Lexemes	1000 - Lexemes	Sum
Kryukov	589	411	1000
Quiet Don	646	354	1000
Sum	1235	765	2000

Table 4.16. Kryukov vs. *Quiet Don* lexeme counts

- H : Kryukov could have written *Quiet Don*
- A : Kryukov did not write it

find the P-value.

For a more recent analysis, which decides strongly in Sholokhov's favor, read the 2007 paper by Nils Lid Hjort, 'And quiet does not flow the Don' at `https://www.mn.uio.no/math/english/research/projects/focustat/news-and-events/hjort_consilience.pdf`.

4.10 Fishing Laws

Three countries, Anchuria, Celophania, and Sikinia, lie on the shores of a lake. The weights of the fish in the lake are uniformly distributed between 0 and 1, which means that the weight of the next fish caught can be conveniently determined by a call to random.random(). The inhabitants of these countries are passionate anglers, so strict laws are enforced to conserve the fish population. Fishing is allowed once a week, and each nation imposes its own rule A, C, and S, defined below, which determines when an angler must stop fishing.

Let $W_1, W_2, W_3, ...$ be the weights of the successive fish caught by an angler. The stopping rules state:

- A: An angler must stop when $W_{n-1} < W_n$.
- C: An angler must stop when $W_1 + W_2 + ... + W_n > 1$.
- S: An angler must stop when $W_n > W_1$.

The number X of fish caught by an angler is a random variable in the range $2, 3, 4, ...$

The probabilities that an angler's catch consists of *exactly n or more* fish, are defined as:

$$p_n = P(X = n),$$
$$q_n = P(X > n).$$

$E = E(X)$ is the expected number of fish in a catch.

We want to investigate each country's p_n and q_n fishing distributions, and their $E(X)$ catch expectation. We'll outline how to do this for Anchuria through

simulation (using the 'A' stopping rule), and leave the investigation of Celophania and Sikinia to you.

```
catchFreq = [0]*(MAX_FISH+1)
totFish = 0
for i in range(NUM_ANGLERS):
  prec = random.random()
  next = random.random()
  numFish = 2  # catch at least two
  while prec >= next:
    prec = next
    next = random.random()
    numFish += 1
  catchFreq[numFish] += 1
  totFish += numFish
print("  n      p         q     ")
q = 1
for i in range(2,MAX_FISH+1):
  if catchFreq[i] > 0:
    p = catchFreq[i]/NUM_ANGLERS
    q -= p
    print(f"{i:3} {p:9.5f} {q:9.5f}")
print(f"E = {totFish/NUM_ANGLERS:.5f}")
```

Listing 4.17. Simulating Anchuria's fishing

Typical output:

```
> python Anchuria.py
  n      p         q
  2   0.50229   0.49771
  3   0.33217   0.16554
  4   0.12436   0.04118
  5   0.03354   0.00764
  6   0.00652   0.00112
  7   0.00097   0.00015
  8   0.00013   0.00002
  9   0.00002  -0.00000
E = 2.71336
```

Listing 4.17 (Anchuria.py) utilizes `prec` and `next` to stand for the preceding and next fish caught, and `catchFreq` holds the frequencies of different size catches.

Based on this output, we guess that $q_n = \frac{1}{n!}$, while E seems very close to $e = 2.71828182$.

Exercises

(1) Write the corresponding program for rule C.

(2) Write the corresponding program for rule S.

The output from your 'C' program should be quite similar to A's results. However, for the data produced by rule S, it's fairly easy to guess q_n and p_n, but not the expectation. For instance, when we ran our 'S' program ten times, the reported E values were 10.84, 46.56, 90.92, 10.60, 214.85, 132.89, 22.06, 14.44, 47.59, and 26.27.

4.10.0.1 Math to the Rescue. It's time to prove our observations.

For $n \geq 2$, $p_n = P(X = n)$ and $q_n = P(X > n)$. This implies that $q_n = 1 - \sum_{i=2}^{n} p_i$, and that $p_n = q_{n-1} - q_n$.

For rule A, $q_n = X > n$ is the situation when $W_1 > W_2 > ... > W_n$. The probability of such a decreasing sequence of length n is $\frac{1}{n!}$. Thus

$$q_n = \frac{1}{n!}, \quad p_n = q_{n-1} - q_n = \frac{n-1}{n!}, \quad np_n = \frac{1}{(n-2)!},$$
$$E(X) = \sum_{n\geq 2} \frac{1}{(n-2)!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + ... = e.$$

Now let's tackle rule S. Clearly $X > n$ when $W_1 = \max(W_1, ..., W_n)$, and the probability of this is $q_n = 1/n$. Thus

$$p_n = q_{n-1} - q_n = \frac{1}{n(n-1)}, \quad np_n = \frac{1}{n-1}, \quad n = 2, 3, 4, ...$$

and

$$E(X) = \sum_{n\geq 2} np_n = 1 + \frac{1}{2} + \frac{1}{3} + ...,$$

which is the harmonic series, and so infinite. This explains why the reported expectations for our 'S' code were so erratic – we were trying to approximate infinity.

Finally, rule C. The probability $q_n = W_1 + ... + W_n \leq 1$ can be treated as the volume of an n-dimensional tetrahedron. The following reasoning shows that the volume V_n of this shape is $\frac{1}{n!}$.

Cavalieri's Principle (`https://en.wikipedia.org/wiki/Cavalieri%27s_principle`) implies that the volume of an n-dimensional pyramid is C_n(base)(height), where the constant C_n depends only on n. We determine C_n by decomposing a cube with unit length edges into $2n$ pyramids with the $2n$ faces as their bases and the center of the cube as the apex vertex of all the pyramids. This means that $2nC_n\frac{1}{2}1 = 1$ and hence that $C_n = \frac{1}{n}$.

This is probably easiest to visualize in 3D (i.e. when $n = 3$). Fig. 4.13 illustrates how six ($2n$) square pyramids fit together to form a cube, with their bases as the cube's six sides.

Fig. 4.13 comes from the website 'The Volume of a Hyperpyramid' (`http://physicsinsights.org/pyramids-1.html`) which does a nice job of explaining how the volume of an n-dimensional square pyramid is calculated.

The points satisfying $W_1 + ... + W_n \leq 1$ can be treated as an n-dimensional hyperpyramid with the W_n vertex as its apex and the other $(n-1)$-points forming its base. The base coordinates also satisfy $W_1 + ... + W_{n-1} \leq 1$ which means that they define a $(n-1)$-dimensional hyperpyramid. This is easiest to understand by thinking of a 3D tetrahedron; its base is a triangle, a 2D hyperpyramid.

This recursive nature for hyperpyramids leads to a recursive definition for their volume: $V_n = \frac{1}{n}V_{n-1}$. This and $V_1 = 1$ imply that $V_n = \frac{1}{n!}$.

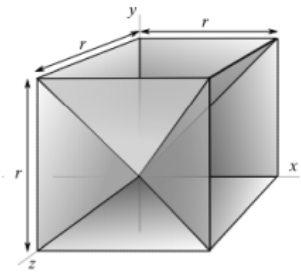


Figure 4.13. Six square pyramids forming a cube

Exercise

Antithetic variables. If U_1 is uniformly distributed in (0,1), then $U_2 = 1 - U_1$ has the same distribution. Suppose the count X in rule C is based on U_1 and Y is based on U_2. If X is below its expected value then Y will be above. The average $(X + Y)/2$ is also an estimate of e with a lower spread (or *variance*). Write a program which uses this idea to estimate e; it helps to realize that min(X,Y) always equals 2.

4.11 Probability of Not Being Lonesome

Let $M = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ be a set of ten people. Each person randomly chooses two of the others as friends. A person chosen by nobody will be deemed "lonesome" (even if other people are his friends). Find the expected number of lonesome people, and the probability that nobody is lonesome.

lonesome.py performs friend generation multiple times, and tallies the results.

Every person is given two friends, who are recorded in the friends list. After everyone has been 'friended', the list is examined for anyone who is still not a friend of someone else, and the `numLone` counter is incremented. Alternatively, if there are no lonely people in the list then `numNotLone` is incremented.

Typical output:

```
> python.exe lonesome.py
Not Lonesome = 2689 ; Lonesome = 10546
Prob Not Lonesome = 0.2689
Expected No. Lonesome = 1.0546
```

We can easily derive the expected number of lonesome people, but formulating the probability of not being lonesome is messy.

Consider any person, say #0. Any other person does not choose #0 with probability $(8/9)(7/8) = 7/9$. Therefore none of the nine others chooses #0 with probability $(7/9)^9$. The expected number E of lonesome people is np with $n = 10$ and $p = (7/9)^9$, so $E = 10 * (7/9) \approx 1.041597$.

The number of choices which leave nobody lonesome can be expressed by means of the "inclusion-exclusion principle" [**Niv65**]. We find P(nobody is lonesome) = 0.274293753....

4.12 Runs

The binary sequence

$$W = 10101101001011000101100101011000101010011001010100$$

consists of 36 runs (i.e. blocks of successive equal digits), made up of 18 runs of ones and 18 runs of zeros. It was concocted by a student to represent 50 coin tosses without bothering to use a coin, and appears to be realistic – it is quite irregular, utilizing 23 ones and 27 zeros. Yet a simple test reveals the student to be a miserable "forger", because his sequence has too many runs.

How many runs should we expect in an n-sequence?

Let $E(k)$ be the expected number of runs in a series of k coin tosses. $E(k+1)$ will only depend on the value of the previous coin toss k and and the current one, $k+1$. If the k-th and $k+1$-th tosses are different, then $E(k+1)$ will increase by 1. We translate this behavior into an equation:

$$E(k+1) = p(pE(k) + (1-p)(1+E(k)) + (1-p)((1-p)E(k) + p(1+E(k)))$$

where p is the probability of a head (or 1 or 'H'). The expression represents the four possibilities for the k-th and $k+1$-th tosses (i.e. HH, HT, TT, TH).

Expanding, and rearranging, gives

$$E(k+1) = 2p(1-p) + E(k)$$

The initial coin toss always starts a run, so

$$E(1) = 1$$

This allows us to rewrite the recursive definition as

$$E(k+1) = 2p(1-p)k + 1$$

or

$$E(n) = 1 + 2(n-1)p(1-p)$$

If our coin is fair, then $p = 1/2$, so

$$E(n) = 1 + (n-1)/2 = (n+1)/2.$$

For $n = 50$ we have $E(n) = 25.5$, but the W sequence has $R = 36$ runs. What's the probability of R $\geq$ 36 with $n = 50$ tosses? Listing 4.18 (runtest.py) finds an answer by simulation.

```
NUM_TRIALS = 10000

n, runs = map(int, input("n runs=? ").
    split())
count = 0
for i in range(NUM_TRIALS):
  numRuns = 1
  prevFlip = random.choice(['H', 'T'])
  for j in range(n-1):
    flip = random.choice(['H', 'T'])
    if flip != prevFlip:
      numRuns += 1
    prevFlip = flip
  if numRuns >= runs:
    count += 1
print(f"Prob of Runs >= {runs} = {(
    count/NUM_TRIALS):.5f}")
```

Listing 4.18. Probability for run numbers

A few different outputs for $n = 50$ and runs $= 36$:

```
> python.exe runtest.py
n runs=? 50 36
Prob of Runs >= 36 = 0.00200

> python.exe runtest.py
n runs=? 50 36
Prob of Runs >= 36 = 0.00230

> python.exe runtest.py
n runs=? 50 36
Prob of Runs >= 36 = 0.00120
```

In other words, the 36 runs in the student's W sequence is very unlikely.

The code repeats the experiment multiple times, each time generating n bits and totaling the coin changes. If the total is greater than our expected run count then a counter is incremented, which is used to generate a probability $P(R \geq$ runs) at the end.

As an alternative to Listing 4.18 we could have reused Listing 4.15. We rephrase the problem as "What's the probability of getting 35 or more successes in 49 tosses of a fair coin?", which becomes a call to binomialSum.py with $n = 49, p = 0.5, c = 0, d = 14$:

```
> python.exe binomialSum.py
n p c d=? 49 0.5 0 14
0.00190083
```

A similar result.

4.12.1 The Expected Length of the Longest Run. This time we'll generate n random bits one by one and keep track of the length of the longest run.

Let's consider each run of heads in a sequence as the number of consecutive heads beginning with the first toss or immediately after a tail occurs. In total, there will be approximately nq head runs, the expected number of tails.

Around nqp of these head runs will contain at least one head, nqp^2 will be at least two heads long, and so on. As long as nqp^x is greater than or equal to one, at least one run of length x or more can be expected. Eventually for larger values of x, nqp^x will fall below 1, which means that getting a run as long as x is unlikely. Hence we should find ER_n, the expected length of the longest head run, when $nqp^{ER} \approx 1$. Rearranging, we get $ER_n \approx \log_{1/p}(nq)$. For our fair coin,

$p = 1/2, ER_n \approx \log_2 n - 1$. So for $n = 50$ (the length of W), we would expect ER_n to be near 4.6. A more accurate analysis produces:

$$ER_n = \log_{1/p}(nq) + \gamma/\ln(1/p) - 1/2$$
$$\text{Var } R_n = (\pi^2/6)\ln^2(1/p) + 1/12$$

where $\gamma = 0.5777...$ is Euler's constant, and Var is the variance. With a fair coin, these become

$$ER_n = \log_2(n/2) + \gamma/\ln 2 - 1/2 \approx \log_2 n - 2/3$$
$$\text{Var } R_n = (\pi^2/6)\ln^2 2 + 1/12 \approx 3.5070$$

The variance is remarkable since it's independent of n, so the standard deviation of even a very long run is still only approximately 1.873.

For more details on this analysis, an excellent resource is 'The Longest Run of Heads' by Mark F. Schilling, *The College Mathematics Journal*, Vol. 21, No.3, 1990, pp.196-207 (available at `https://maa.org/sites/default/files/pdf/upload_library/22/Polya/07468342.di020742.02p0021g.pdf`).

Using this improved (but still approximate) equation, when $n = 1024$ we have $ER_n \approx 9.333$, which we test with Listing 4.19 (`maxruntest.py`).

```
n = int(input("n=? "))
maxRuns = []
for j in range(NUM_TRIALS):
  maxRun = 0; count = 0
  while count < n:
    i = 0
    while (random.choice(['H','T']) ==
          'H') and (count < n):
      i += 1; count += 1
    if i > maxRun:
      maxRun = i
  print(f"{maxRun:2}", end=' ')
  if (j+1)%20 == 1:
    print()
  maxRuns.append(maxRun)
print()
print(f"Mean maxrun = {statistics.mean
    (maxRuns):.2f}")
print(f"Stdev = {statistics.stdev(
    maxRuns):.2f}")
```

Listing 4.19. Length of the longest run

A typical run:

```
> python.exe maxruntest.py
n=? 1024
10
10  8 10 14  9  8  9  8 12 14
 8  8 10 12 15  8 12  8 10  9
  : # more lines, not shown
 8 10 10 11  9 16  9 13 13 12
11  9 13 10 13  9 12  8  8
Mean maxrun = 10.35
Stdev = 1.91
```

4.12.2 Time on the Winning Side. Contrary to popular notions, it's quite likely that in a long coin-tossing game one of the players remains on the winning

side practically the whole time. By winning we mean that the player's total number of heads (or tails) is almost always greater than their opponent's number of tails (or heads).

Most people would say that the time spent winning would be shared roughly equally. This seems to fit with the law of large numbers which states that as the number of trials increases, the proportion of heads vs. tails will converge on the expected value of 0.5.

largeFlips.py confirms this latter point (see the left hand chart in Fig. 4.14). However, it also plots the absolute difference between the numbers of heads and tails which does *not* remain close to 0.

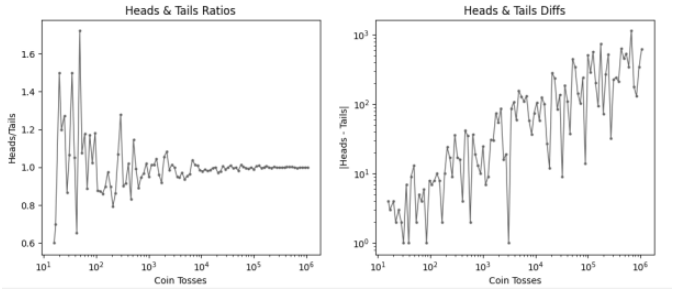


Figure 4.14. Plotting head/tail ratio and head/tail difference

The printed output reinforces the graphical results:

```
> python largeFlips.py
Final abs diff: 1018
Final ratio: 1.001943567802451
```

The 'Heads and Tails Diffs' graph in Fig. 4.14 uses logarithmic scales to highlight the linear relationship between the axes, which means there's a power relationship in the original data. One way to understand this is to think about coin tosses as a 1-dimensional random walk, starting from the origin. A 'H' moves the position one step to the right, and 'T' moves one step to the left. Since forward and backward steps are equally likely, the expected average finishing position must be back at the origin, but the interesting question is how far away from the origin, on average, we might expect to reach, regardless of direction.

We can ignore the direction issue by computing the expected value of the *square* of the distances from the origin, then take its square root, the "root mean square" or rms distance. In general, this distance from the origin, after a random walk of n steps, will be $\sqrt{n}$, and is the power law displayed in the 'Heads and Tails Diff' chart. As the graph shows, after 1 million coin tosses, the absolute difference between the number of heads and tails is on average about 1000.

This difference between the number of heads and tails grows as the number of coin tosses increases, which means that a player who gains a slight advantage over their opponent in terms of their number of heads (or tails) is likely to see that advantage grow.

headsDiffs.py visualizes what this growing advantage can mean by generating data for the number of heads - number of tails (i.e. the head's advantage) over a long series of coin flips, as the graph in Fig. 4.15.

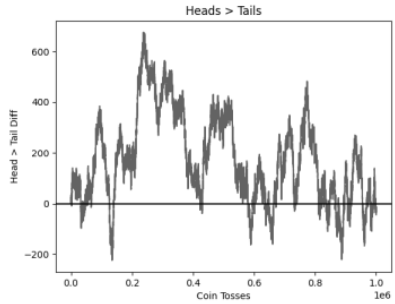


Figure 4.15. Plotting heads > tails

The program also calculates the percentage time (in terms of the fraction of the total number of coin tosses) that heads is in the lead. The following shows a few of the outputs for different runs:

```
> python headsDiffs.py
Final heads > tails diff: -44
% Times that Heads > Tails:
Pos: 84.02%; Neg: 15.98%

> python headsDiffs.py
Final heads > tails diff: 258
Pos: 82.13%; Neg: 17.87%

> python headsDiffs.py
Final heads > tails diff: 632
Pos: 59.53%; Neg: 40.47%

> python headsDiffs.py
Final heads > tails diff: 746
Pos: 19.29%; Neg: 80.71%

> python headsDiffs.py
Final heads > tails diff: 548
Pos: 25.30%; Neg: 74.70%
```

Only one of the five runs reports anything near an equal balance between the numbers of heads and tails during the game, while two favored heads, and two favored tails.

headsLeads.py automates the repeated calls to headsDiffs.py (10000 times) and stores the heads lead over the tails (if there is one) for each trial. After all the runs, the number of occurrences of each lead value is tallied and displayed in a bar chart. Fig. 4.16 displays the distribution of the heads lead.

The graph shows that the player collecting heads will most probably find himself in one of two situations – miserably far behind the player collecting tails (this corresponds to the left-hand side of Fig. 4.16) or happily way in front of his opponent (that's the right-hand side). Most surprisingly, the least likely situation is when the two players have about the same score. Fig. 4.16 displays data for 80 flips/trial, so the 'balanced' case is when head has a 'lead' of 40, which means that the tails player has a similar lead.

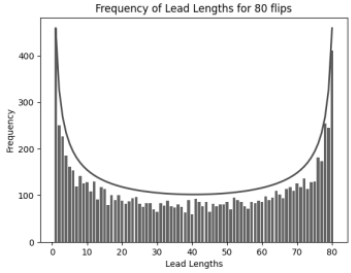


Figure 4.16. The heads lead distribution

The same kind of graph will be produced if headsLeads.py is modified to record the tails lead rather than the heads (simply change the if-test that checks the random choice from 'H' to 'T'). This reinforces the fact that the coin is fair, since both players have the same lead distribution.

In conclusion, it's quite likely that in a long coin-tossing game one of the players remains on the winning side almost all the time.

The curve drawn over the bar chart in Fig. 4.16 is for the function:

$$f(x) = \frac{1}{\pi\sqrt{x(1-x)}}$$

which fits the lead distribution curve remarkably well. It can be readily integrated to obtain an approximation to the cumulative distribution:

$$P(\frac{L_n}{n} \leq x) = \frac{2}{\pi} \arcsin\sqrt{x}$$

where L_n is the lead after n coin tosses. This is known as the arcsine law or 'Law of Long Leads'. For more information, the best source is Chapter 3 of Feller [**Fel68**]. A great online read is 'What is the Arcsine Law?' by Ethan Ackelsberg at `https://math.osu.edu/sites/math.osu.edu/files/What_is_2018_Arcsine_Law.pdf`.

4.12.3 How Long Before we Get HHHH?. A fair coin with faces 'H' and 'T' is flipped until HHHH occurs in the run. Let X be the number of flips we have to make before this situation arises.

We can represent the sequence of coin tosses using a directed graph (Fig. 4.17) like the ones in Appendix A. If the sequence produces a 'T' then the graph traversal returns to 'Start'.

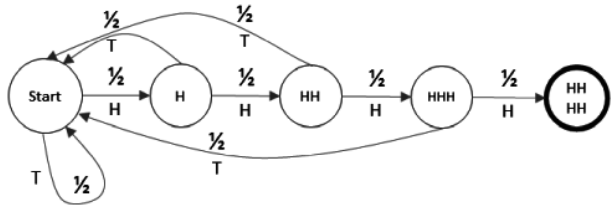


Figure 4.17. Transition probabilities for getting to HHHH

We want to find the probability P_n that exactly n flips will be required to reach HHHH:

$$(i) \quad P_n = P(X = n), \quad (ii) \quad E(X) = \sum_{n \geq 1} n * p_n.$$

Fig. 4.17 shows that

$$p_i = 0 \text{ for } i = 1, 2, 3, \quad p_4 = \frac{1}{16}, \quad p_5 = \frac{1}{32},$$

If we assume p_n is the success state (HHHH), reached after n tosses, then the preceding states HHH, HH, and H are p_{n-1}, p_{n-2}, and p_{n-3} respectively. Therefore, the graph structure implies that

$$p_n = \frac{1}{2}p_{n-1} + \frac{1}{4}p_{n-2} + \frac{1}{8}p_{n-3} + \frac{1}{16}p_{n-4}, \quad n \geq 5.$$

Creating a new equation by replacing n by $n-1$, and subtracting the two, we get

$$p_n = p_{n-1} - \frac{p_{n-5}}{32}, \quad n \geq 6.$$

This recurrence is also valid for $n = 5$ if we define $p_0 = 1$. Then

$$p_n = p_{n-1} - \frac{p_{n-5}}{32}, \quad n \geq 5.$$

`expectHHHH.py` is an iterative program that uses this equation to find E(X). For $n = 1000$ we get the expected number of flips = 30.0.

4.12.4 Which is More Likely: HH or TH?. When the tossing of two coins is considered as a single event, the probability of getting HH or TH is of course the same, $\frac{1}{4}$. However, if we ask this question in the context of coin toss *runs*, the answer is very different. We now ask:

> Suppose you were to keep flipping a coin until it landed either HH or TH on two consecutive flips. Which of these two sequences would you predict would occur first?

It turns out that TH is three times more likely to appear first. Listing 4.20 (`countPairs.py`) shows how this can be simulated.

```
thCount = 0; hhCount = 0; cnts = []
for j in range(NUM_TRIES):
  prevFlip = random.choice(['H', 'T'])
  while True:
    flip = random.choice(['H', 'T'])
    if prevFlip == 'T' and flip == 'H':
      thCount += 1; break
    elif prevFlip == 'H' and flip == 'H':
      hhCount += 1; break
    else:
      prevFlip = flip
print(f"TH%:{(thCount/NUM_TRIES):.2%};HH%:{(hhCount/NUM_TRIES):.2%}")
```

Listing 4.20. HH vs. TH

The output:

```
> python countPairs.py
TH%: 74.94%; HH%: 25.06%
```

This strange behavior is best understood by considering Fig. 4.18.

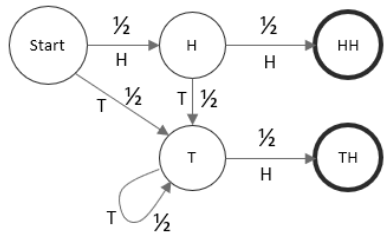


Figure 4.18. Transition probabilities for getting to HH and TH

It's clear that the probability of reaching HH is $\frac{1}{4}$. The probability for reaching TH can be obtained in two stages. The first is to realize that once the player enters the 'T' state, then there's no other possibility but to (eventually) reach TH, and so we can assign a probability of 1 to that state. There are two paths to the 'T' state – in two steps via the 'H' state or directly from 'Start'. Therefore the probability of reaching TH is $\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$.

More complicated (and entertaining) examples of this kind can be found in 'Penney Ante: Counterintuitive Probabilities in Coin Tossing', by R.S. Nickerson, *The UMAP Journal*, 28.4, 2007, pp.503-532.

Exercises

(1) Write a program with inputs n, r which computes the mean waiting time for a sequence with r ones in a row. Devise a formula for the time.

(2) A fair coin with faces 0 and 1 is flipped until the word 1001 occurs for the first time. Let X be the number of flips until that arises. Draw a graph similar to Fig. 4.17 and find

$$p_n = P(X = n) \quad \text{and} \quad E(X) = \sum_{n \geq 1} n * p_n.$$

4.13 Waiting for a Complete Set

Spin the spinner in Fig. 4.19 until all the numbers from 1 to n have occurred.

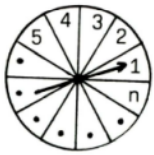


Figure 4.19. n possible values

Let S be the number of spins to complete the set, which is also known as the *coupon collector's problem*. We want to estimate the expected number $E(S)$ by simulation. Listing 4.21 (coupons.py) is the program for this equiprobable case.

```
def collect(n):
  nTries = 0; collected = set()
  while len(collected) < n:
    coupon = random.randint(1, n)
    nTries += 1
    collected.add(coupon)
  return nTries

if __name__ == "__main__":
  n = int(input("n=? "))
  tot = 0
  for i in range(10):
    spins = collect(n)
    print(spins, end = ' ')
    tot += spins
  print()
  print("Avg:", tot/10)
```

Typical output for $n = 6$ and $n = 10$:

```
> python coupons.py
n=? 6
15 13 11 16 15 8 19 15 9 10
Avg: 13.1

> python coupons.py
n=? 10
43 27 27 26 19 17 30 22 36 25
Avg: 27.2
```

Listing 4.21. How long to collect everything?

coupons.py always carries out 10 trials for a given n, reporting the total number of spins needed to complete the collection for each trial, and the average.

This type of coupon collection (where each item has equal probability of being selected) is easy to analyze. Problem 2 of Appendix A utilizes a directed graph to show that

$$E(S) = n(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) \approx n \log n. \tag{4.8}$$

For $n = 6$ and $n = 10$ we get $E(S) = 14.7$ and 29.29, respectively.

Another approach uses the fact that the problem has a geometric distribution since the sequence of independent trials (i.e. spins of the pointer), involve only two outcomes (failure or success). If the probability of success on each trial is p, then the probability that the k-th trial is the first success is

$$Pr(X = k) = (1 - p)^{k-1} p$$

We'll use the fact that a geometric random variable X has the following properties:

$$\begin{aligned} E(X) &= 1/p \\ \text{Var}(X) &= (1 - p)/p^2 \end{aligned}$$

Let s_i be the spin that collects the i-th coupon after $i - 1$ coupons have been obtained already. Then $S = s_1 + \cdots + s_n$, and the probability of collecting a new coupon is

$$p_i = \frac{n - (i - 1)}{n}.$$

Therefore, s_i has a geometric distribution with expectation

$$\frac{1}{p_i} = \frac{n}{n - i + 1}.$$

By the linearity of expectations we have:

$$\begin{aligned} E(S) &= E(s_1 + s_2 + \cdots + s_n) \\ &= E(s_1) + E(s_2) + \cdots + E(s_n) \\ &= \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= \frac{n}{n} + \frac{n}{n-1} + \cdots + \frac{n}{1} \\ &= n \cdot \left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{1}\right) \\ &= n \cdot H_n. \end{aligned}$$

H_n is the n-th harmonic number, such that:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.$$

Using H_n's asymptotics:

$$E(S) = n \cdot H_n \approx n \log n + \gamma n + \frac{1}{2} \tag{4.9}$$

where $\gamma = 0.5772...$ is the Euler constant. This confirms that the expected number of trials needed to complete a set of size n is $O(n \log(n))$ as we saw in Equ. (4.8).

The variance is determined in a similar way:

$$\begin{aligned}
\mathrm{Var}(S) &= \mathrm{Var}(s_1 + s_2 + \cdots + s_n) \\
&= \mathrm{Var}(s_1) + \mathrm{Var}(s_2) + \cdots + \mathrm{Var}(s_n) \\
&= \frac{1-p_1}{p_1^2} + \frac{1-p_2}{p_2^2} + \cdots + \frac{1-p_n}{p_n^2} \\
&< \left(\frac{n^2}{n^2} + \frac{n^2}{(n-1)^2} + \cdots + \frac{n^2}{1^2}\right) \\
&= n^2 \cdot \left(\frac{1}{n^2} + \frac{1}{(n-1)^2} + \cdots + \frac{1}{1^2}\right) \\
&< \frac{\pi^2}{6} n^2
\end{aligned}$$

since

$$\frac{\pi^2}{6} = \frac{1}{1^2} + \frac{1}{2^2} + \cdots + \frac{1}{n^2} + \cdots,$$

the solution to the *Basel Problem* (see Sec. 6.7.7).

`couponsHarm.py` tests whether $E(S) = n \cdot H_n$ by running each coupon collection trial between $n = 1$ and $n = 100$ a hundred times, and plotting the average against the curve for $n \cdot H_n$.

Fig. 4.20 confirms that the expected values do closely follow the $n \cdot H_n$ curve.

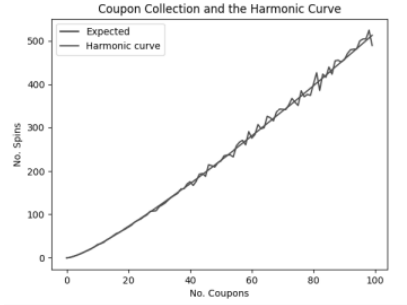


Figure 4.20. Coupon collection is harmonious

4.13.1 Unequal Probabilities. The case of unequal probabilities for obtaining different coupons is more complicated. It's still possible to give a closed formula similar to the one in Equ. (4.8), but the formula size grows proportionally with $n!$. To show you what this means we've calculated the exact formulas for the spinners in Fig. 4.21, where $n = 3$ and $n = 4$.

The relations $z + y + ... = 1$ could be used to eliminate one of the variables but the formulas are simpler if we don't do that.

For Fig. 4.21a, with $n = 3$, we get for $E(S)$

$$1 + \frac{x}{1+x}(1 + \frac{y}{z} + \frac{z}{y}) + \frac{y}{1-y}(1 + \frac{x}{z} + \frac{z}{x}) + \frac{z}{1-z}(1 + \frac{x}{y} + \frac{y}{x}),$$

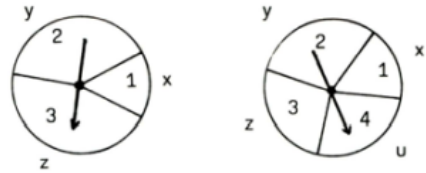


Figure 4.21. More complex coupon collection

For Fig. 4.21b, with four probabilities x, y, z and u, we get for $E(S)$

$$1+\frac{x}{1-x}\left[1+\frac{y}{z+u}(1+\frac{u}{z}+\frac{z}{u})+\frac{z}{y+u}(1+\frac{y}{u}+\frac{u}{y})+\frac{u}{y+z}(1+\frac{y}{z}+\frac{z}{y})\right]$$
$$+\frac{y}{1-y}\left[1+\frac{z}{u+x}(1+\frac{x}{u}+\frac{u}{x})+\frac{u}{z+x}(1+\frac{z}{x}+\frac{x}{z})+\frac{x}{z+u}(1+\frac{z}{u}+\frac{u}{z})\right]$$
$$+\frac{z}{1-z}\left[1+\frac{u}{x+y}(1+\frac{y}{x}+\frac{x}{y})+\frac{x}{u+y}(1+\frac{u}{y}+\frac{y}{u})+\frac{y}{u+x}(1+\frac{u}{x}+\frac{x}{u})\right]$$
$$+\frac{u}{1-u}\left[1+\frac{x}{y+z}(1+\frac{z}{y}+\frac{y}{z})+\frac{y}{x+z}(1+\frac{x}{z}+\frac{z}{x})+\frac{z}{x+y}(1+\frac{x}{y}+\frac{y}{x})\right]$$

For $n = 10$ the exact formula would fill hundreds of volumes, and would be of no practical use. Simulation is the only feasible approach for obtaining numerical results for these problems.

Exercises

(1) *Collecting two complete sets.* A spinner can land on $1, 2, ..., n$ with equal probability. What is the expected number of spins until all the numbers from 1 to n have occurred at least twice? Run a simulation for $n = 6$ and $n = 10$.

(2) Extend Listing 4.21 so that it generates 100 complete sets and calculates the mean number of spins. Run the program for $n = 6$ and $n = 10$ and compare the results with the exact values 14.7 and 29.29.

(3) Write a program for the complete set problem where you can vary both the outcome i and its probability p_i. Run it for $n = 10$ where $p_i = \frac{1}{i} - \frac{1}{i+1}$ for $i = 1, 2, ..., 9$, but $p_{10} = \frac{1}{10}$.

(4) Rooks (castles) are placed on the squares of an $n \times n$ chessboard one at a time and with the choice of any square the same $1/n^2$. This continues until all the squares on the board are threatened by the rooks. This version allows the repeated choice of a square, so it's possible that a square might end up containing several rooks.

Find the expected number of rooks when the program is run with $n = 8$. Theory predicts that $E_M \approx 17.12865$.

(5) (Continuation.) Suppose now that each rook placement is only permitted in an unoccupied square. Calculate the expected number of rooks needed to threaten all the squares. Theory predicts $E_M \approx 15.0029$.

(6) *A total tour of a regular n-gon.* A beetle starts at a vertex of a regular n-gon and goes for a random walk along its edges. Find by simulation the expected time for the bug to complete a total tour of all the edges. (The beetle completes one edge in one time unit.) *Hint*: consider the $n = 3, 4, 5, ...$ cases in turn, and then guess a formula.

(7) Start with n coins. Toss all the coins at once, and eliminate those that land showing a head. Find by simulation the expected number E(n) of tosses until all the coins have been eliminated. You should get the following results: $E(3) = 22/7$, $E(4) = 368/105$, $E(15) = 5.287$. Also have a look at Ex. 2 of Sec. 5.4.

4.14 The Birthday Problem (and Factoring)

The birthday problem asks for the probability that in a set of n randomly chosen people at least two will share a birthday. The birthday paradox refers to the counterintuitive fact that only 23 people are needed for that probability to exceed 50%.

Given n people, the probability P_n, that there are *no* common birthdays among them is

$$P_n = \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right)\cdots\left(1 - \frac{n-1}{365}\right)$$

The first factor is the probability that two given people do not have the same birthday. The second factor is the probability that a third person does not have a birthday in common with either of the first two. This continues until the last factor is the probability that the n-th person does not have a birthday in common with any of the other $n - 1$ people.

We want $P_n < 1/2$, which can be determined by examining the output from Listing 4.22 (birthday.py).

```
NUM_PEOPLE = 30

prob = 1
for i in range(1, NUM_PEOPLE+1):
  prob *= (1-((i-1)/365))
  print(f"p({i:2})={prob:.4f} ", end = ' ')
  if i%4 == 0:
    print()
```

Listing 4.22. Calculating P_n up to $n = 30$

The output:

```
> python birthday.py
p( 1)=1.0000  p( 2)=0.9973  p( 3)=0.9918  p( 4)=0.9836
p( 5)=0.9729  p( 6)=0.9595  p( 7)=0.9438  p( 8)=0.9257
p( 9)=0.9054  p(10)=0.8831  p(11)=0.8589  p(12)=0.8330
p(13)=0.8056  p(14)=0.7769  p(15)=0.7471  p(16)=0.7164
p(17)=0.6850  p(18)=0.6531  p(19)=0.6209  p(20)=0.5886
p(21)=0.5563  p(22)=0.5243  p(23)=0.4927  p(24)=0.4617
p(25)=0.4313  p(26)=0.4018  p(27)=0.3731  p(28)=0.3455
p(29)=0.3190  p(30)=0.2937
```

There must be at least 23 people in a room in order for the odds to favor at least two of them having the same birthday.

This answer is much smaller than most people expect. Even for $n = 30$, where the odds of a common birthday increase to 70.6%, most people still find it hard to believe.

One reason may be that they are asking themselves a different question, namely, "How many people need to be present for there to be a 50% chance that someone else has *my* birthday?" The answer to that question *is much* larger than 23. The probability that no one out of n people has a birthday on a given day is $(1 - 1/365)^n$. If we want a value for n where the probability exceeds 0.5 that at least one other person has your birthday:

$$\begin{aligned}(1 - \tfrac{1}{365})^n &\geq \tfrac{1}{2} \\ n \cdot \log(\tfrac{364}{365}) &\geq \log(\tfrac{1}{2}) \\ n &\geq 252.65\end{aligned}$$

We can generalize the birthday problem to be about finding n people who share a birthday over N different days:

$$P_n = \left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\cdots\left(1 - \frac{n-1}{N}\right)$$

It will be useful later to have a simple approximation which links n to N. Given a large number of people, n, there are $\binom{n}{2} = \frac{n(n-1)}{2} \approx n^2/2$ pairs. The probability that a given pair has the same birthday is $1/N$, so the probability that they do not have the same birthday is $1 - (1/N)$. Therefore, the probability that no pair has a common birthday is

$$P_n = \left(1 - \frac{1}{N}\right)^{n^2/2} \approx \left(1 - \frac{1}{N}\right)^{N(n^2/2N)} \approx e^{-n^2/(2N)} \tag{4.10}$$

The last step is utilizing Euler's product limit formula for e^x:

$$e^x = \lim_{n\to\infty}(1+\frac{x}{n})^n$$

When $P_n = 1/2$, we can rearrange Equ. (4.10) to give

$$n \approx \sqrt{2\ln 2}\sqrt{N}$$

4.14.1 The Expected Waiting Time. Let E(N) be the expected waiting time until we get a double birthday. Let $q(N,n)$ be the probability that all n people chosen at random have different birthdays in a 'year' of N days. Therefore:

$$\begin{aligned} q(N,n) &= (1-\tfrac{1}{N})(1-\tfrac{2}{N})\cdots(1-\tfrac{n-1}{N}) \\ &= \tfrac{N}{N}\tfrac{N-1}{N}\tfrac{N-2}{N}\cdots\tfrac{N-n+1}{N} \\ &= \tfrac{1}{N^n}(N(N-1)(N-2)\cdots(N-n+1)) \\ &= \tfrac{1}{N^n}\tfrac{N!}{(N-n)!} \end{aligned}$$

The expected waiting time is the sum of these probabilities for all possibly values of n:

$$E(N) = \sum_{n=1}^{N} q(N,n).$$

This can also be phrased as the average number of people required before we find a pair with the same birthday. In other words, we are calculating the mean of the distribution, as opposed to the earlier median. This averaging problem is relevant to several hashing algorithms explained in Section 6.4, Vol. 3 of Knuth [**Knu97**].

The right hand side of the equation is very close to Srinivasa Ramanujan's Q formula:

$$Q(M) = \sum_{k=1}^{M} \frac{M!}{(M-k)!M^k}$$

Ramanujan found its asymptotic expansion to be:

$$Q(M) \sim \sqrt{\frac{\pi M}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2M}} - \frac{4}{135M} + \cdots$$

A derivation of this asymptote using only elementary calculus is given in Section 1.2.11.3, Vol. 1 of Knuth [**Knu97**]. It can be plugged into E(N), and simplified:

$$E(N) \sim \sqrt{\frac{\pi N}{2}} - \frac{1}{3}.$$

Listing 4.23 (birthexp.py) calculates $E(N)$ in two ways – by summing $q(N,n)$ probabilities from n=1 to N, and by calculating Ramanujan's asymptote.

```
N = int(input("days=? "))
prob = 1
tot = 0
for i in range(1, N+1):
  prob *= (1-((i-1)/N))
  tot += prob
print(f"Mean: {tot:.6f}, Q: {math.sqrt(math.pi*N/2)-1/3:.6f}")
```

Listing 4.23. Computing E(N) and $\sqrt{\pi N/2}$ - 1/3

The output when the number of days is 365.

```
> python birthexp.py
days=? 365
Mean: 23.616586, Q: 23.611200
```

The average number of people required to find a pair with the same birthday is close to the earlier calculation of the number giving a 50% chance for a pair.

4.14.2 An O($n^{1/4}$) Method of Factoring. Aside from the birthday problem's use in hashing, it's also employed in John Pollard's rho algorithm for fast integer factorization. The code's expected running time is proportional to the square root of the smallest prime factor of the composite being factorized. Finding this prime can be implemented in O($n^{1/2}$) running time, where n is the composite, so it is sometimes claimed that the algorithm is O($n^{1/4}$), although there's no rigorous proof of that speed.

As we'll see, the procedure uses a recurrence formula to generate random numbers, but its randomness is far from certain, so neither its running time nor its success are guaranteed. Nevertheless the procedure is highly effective in practice, and also requires a very small amount of computer memory.

We've already looked at factoring using trial division (see Listing 2.13.8). factor.py is a version of that code which returns its values using `yield` rather than by constructing a list. This means that factorsTime() can report the times taken to find each factor, which allow factorsY() to be more fairly compared to our rho() function (which only returns one factor).

Example runs:

```
> python factor.py
n=? 5166523324060302275610181
141325637  in 12.43 secs
36557580306963713  in 3.81 secs
  Total: 16.24 secs

> python factor.py
n=? 280846283204599997
265371653  in 21.86 secs
```

```
1058313049  in 0.00 secs
  Total: 21.86 secs
```

One way to double-check results for larger factorization is via the calculator at `https://www.numberempire.com/numberfactorizer.php`.

4.14.2.1 Applying the Birthday Problem to Factoring. If we are using trial division to search a number $N = p * q$ for its two prime factors, p and q, then the code may have to examine $\sqrt{N}$ numbers before finding one of them. But instead of examining one number at a time, we can pick k: $x_1, x_2, ...x_k$, and check whether any of the pair differences, $x_i - x_j$, divide N. This change parallels the difference between picking one person and asking if their birthday falls on a specific day and picking k people and asking if any two of them share a birthday.

We also know that for a k size of around $\sqrt{N}$, we have even odds of finding a prime. Very roughly speaking, we improve the chances from $1/\sqrt{N}$ with trial division to $1/\sqrt[4]{x}$ with the birthday technique. But there's a downside: if we select k people, then our code will have to perform roughly k^2 pairwise comparisons which is costly in time.

4.14.2.2 GCD to the Rescue. The solution to the pairwise problem is to compare the numbers in a way that increases the chance of a match. Instead of asking if $x_i - x_j$ divides N, we can ask if $\gcd(x_i - x_j, N) > 1$. In other words, we ask if $x_i - x_j$ and N have a non-trivial factor in common. This at once increases the number of chances for successes.

If we ask how many numbers divide N, there are only two: p and q. If we ask how many numbers have $\gcd(x, N) > 1$, we have quite a few more: $p, 2p, 3p.4p, 5p, ...(q-1)p$ and $q, 2q, 3q, ...(p-1)q$. A total of $p + q + 2$ numbers.

So the rho() function will do the following:

- Pick k numbers $x_1, ..., x_k$ at random from between 2 and $N - 1$.
- Test if $\gcd(x_i - x_j, N) > 1$. If it is greater, then $\gcd(x_i - x_j, N)$ is a factor of N (a multiple of p or q).

4.14.2.3 Space for k Numbers. This approach requires the generation of k random numbers all at once. For us to have a 50% chance of success the size of k should be around $N^{1/4}$, requiring a lot of storage. For example, if we're factoring $N \sim 10^{32}$, then we need to examine a range of numbers up to 10^{16} ($N^{1/2}$) which means generating $k = 10^8$ numbers ($N^{1/4}$).

This problem is addressed by *not* generating all k of them at once. The next best thing is to generate them randomly one by one and check consecutive numbers. We keep doing this for a while and *hopefully* we'll get lucky. (This is where the heuristic nature of the rho() function appears.)

We use a function $f()$ that we know generates pseudo-random numbers, such as $f(x) = x^2 + a \bmod N$. We start with a random seed, such as $x_1 = 2$, and keep calling $f(x)$ recursively to generate more x's: $x_2 = f(x_1)$, $x_3 = f(x_2)$, ... The general rule is $x_{n+1} = f(x_n)$.

This works well most of the time, but what about the output from the following $f()$:

```
2, 10, 16, 23, 29, 13, 16, 23, 29, 13, ...
```

In this case, the numbers quickly start cycling and will never find a factor. Indeed, cycling will be an issue with every PRNG eventually, and our rho() must deal with it.

One approach is to revisit Floyd's cycle detection algorithm from Sec. 3.14. The idea is to utilize two pointers A and B, but with pointer B moving twice as fast as A. They start off at the same place and when B overtakes A, we know that B has cycled around at least once.

4.14.2.4 Implementing rho(). Listing 4.24 (pollardRho.py) implements the features we've described: the birthday algorithm, the use of gcd() to increase the likelihood of success, the utilization of a PRNG to produce multiple random numbers, and Floyd's cycle detection.

```
def rho(n):
  if (n <= 1):
    return n
  if (n % 2 == 0):
    return 2
  x = random.randint(2, n-1)  # randoms in [2,n)
  y = x
  c = random.randint(1, n)
  d = 1
  # loop until the prime factor isn't obtained.
  # If n is prime, return n
  while (d == 1):
    x = (modPow(x,2,n) + c+n)%n  # slow move
    y = (modPow(y,2,n) + c+n)%n  # fast move
    y = (modPow(y,2,n) + c+n)%n
    d = math.gcd(abs(x-y), n)
    if (d == n):   # detected a cycle
      print("Retrying")
      return rho(n)
  return d  # 1 < d < n, so found a prime
```

Listing 4.24. Pollard's Rho function

The d variable returned by gcd() is used in three different ways. If $d = 1$ then the number tested against n is co-prime, so not divisible, which means that the search should continue. If $d = n$ then it means that $x - y$ is a multiple of n. This

implies that y (the fast pointer) is at the same position as x (the slow pointer) but after going round a cycle (of length n). We have entered a cycle, which means that we need to restart the function with a new x and c.

millerRabin.py contains a standard implementation of the function $(x^e) \mod m$.

If we test rho() on the composites passed to factor.py in the examples above, the speed-up is quite significant:

```
> python pollardRho.py
n=? 516652332406030227561018 1
One divisor for 516652332406030227561018 1 = 141325637
in 0.10 secs

> python pollardRho.py
n=? 280846283204599997
One divisor for 280846283204599997 = 265371653
in 0.12 secs
```

The original reference for this algorithm is Pollard, J. M. (1975), 'A Monte Carlo method for factorization', *BIT Numerical Mathematics*, 15 (3), pp.331-334. Improvements were later suggested in Brent, Richard P. (1980), 'An Improved Monte Carlo Factorization Algorithm', *BIT*, 20. pp.176-184.

A good textbook reference is Cormen *et al.* [**CLRS22**]. They discuss integer factorization and Pollard's rho algorithm in section 31.9, which sadly has been removed in the current 4th edition, but it can still be found on their website at `http://mitpress.mit.edu/algorithms`.

Exercise

Collect birthdays at random, until you obtain three birthdays for the same day. Repeat this experiment 10000 times and estimate the expected waiting time.

Additional Exercises for Chapters 3 to 4

(1) Write a program which performs n games of Craps and counts the number of wins. Here are the rules:

Roll two dice and total the points showing. If the total is 7 or 11 you win. If it is 2, 3, or 12 you lose. For any other total, call it the "point" and continue rolling the dice until either the total 7 comes up (a loss) or the point comes up again (a win).

Estimate the probability of a win by repeating the experiment 10000 times. Simple theory shows that the winning probability is $w = 244/495 = 0.492492....$ How many experiments do you need to perform to be 95% sure that $w < 0.5$?

(2) A fair coin is flipped until either the sequence 0011 or 1111 comes up. In the first case Abby wins and in the second case Carl wins. Write a simulation which estimates Abby's winning probability.

(3) *Symmetric Random Walk on a Line.* A particle starts at the origin and jumps one step to the left or to the right each second, with a probability of 0.5. Let D_n be the particle's distance from the origin after n steps. Find $E(D_n^2)$.

(4) A particle starts at the origin O and performs a random walk over the plane. At each step it goes from its current position (x, y) to one of its four neighbors $(x+1, y), (x-1, y), (x, y+1), (x, y-1)$ with probability 1/4. Let D_n be its distance from O after n steps. Find $E(D_n^2)$.

(5) A particle starts at O and takes unit steps. The direction of each step is uniformly distributed between 0 and 2π. Let D_n be its distance from O after n steps. Find $E(D_n^2)$ by simulation.

(6) *Neighbors in Lotto.* Six numbers are drawn at random without replacement from 1..49. Find by simulation the probability that the sample drawn contains at least two consecutive numbers, such as 18 and 19.

(7) *Random Walk in Space.* A particle starts at the origin O and takes unit steps from its current position (x, y, z) to one of its six neighbors $(x \pm 1, y, z)$, $(x, y \pm 1, z)$, $(x, y, z \pm 1)$, each with probability 1/6. Let D_n be its distance from O after n steps. Estimate $E(D_n^2)$ by simulation.

(8) *Random Walk in Space in Random Directions.* A particle in space starts at the origin O and takes unit steps. A unit sphere is projected around its current position P, a random point R is chosen on this sphere, and the next step goes from P to R. Let D_n be the particle's distance from O after n steps. Estimate $E(D_n^2)$ by simulation.

(9) *Maximum Test.* What is the probability that among any three successive base ten random digits the middle digit is strictly larger than any of its neighbors? Generate 10000 successive triples of base ten digits and find the proportion. Use:

a) Python's PRNG random.random();

b) The PRNG in Listing 3.22.

(10) *Poker Test.* Generate 1000 words made up of five random decimal digits, find the proportions of the seven types listed in Table 4.17, and compare them with the given probabilities. (More example poker hands are listed at `https://en.wikipedia.org/wiki/List_of_poker_hands`.)

type	example	probability
all distinct	30862	0.3024
one pair	32082	0.5040
two pairs	02772	0.1080
one triple	96066	0.0720
triple and pair	39399	0.0090
quadruple	80888	0.0045
quintuple	55555	0.0001

Table 4.17. Poker probabilities

(11) Re-read the rules of Craps in Ex. 1. Draw a graph for this game and calculate the winning probability exactly (up to 11 digits).

(12) *It all depends on the information.* If you toss a thumbtack, it will land pointing up (U) or down (D) as in Fig. 4.22.

Let $p = P(D)$ so that when we are given the outcome DDDUDUDDDDDU, we test H: $p = \frac{1}{2}$ versus A: $p > \frac{1}{2}$. But to compute the P-value we need more information about X (the number of D's):

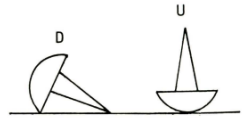


Figure 4.22. Tack positions (D=down; U=up)

a) It was decided in advance to make 12 tosses, so

$$P = P(9 \le X \le 12|H) = 2^{-12}\left(\binom{12}{3} + \binom{12}{2} + \binom{12}{1} + \binom{12}{0}\right) = \frac{299}{4096} \approx 7.3\%$$

b) In this case, the experiment was halted as soon as three U's appeared. Now the results $X = 9, 10, 11, 12, 13, ...$ are as extreme, or more extreme, than what we were given, so we must find

$$P = P(x \ge 9|H) = \sum_{x=9}^{\infty} p(x), \quad p(x) = \binom{x+2}{2} 2^{-x-2}\frac{1}{2} = \binom{x+2}{2} 2^{-x-3}.$$

(i) Find the P-value by simulation.

(ii) Find the exact P-value by avoiding an infinite sum.

(13) The sequence $u_1, ..., u_n$ is *unimodal* if there exists an s with

$$u_1 < u_2 < ... < u_s < u_{s+1} > ... > u_n \quad (s = 1 \text{ or } s = n \text{ are allowed})$$

a) How can we test for unimodality?

b) Let p_n be the probability that a sequence of n randomly chosen real numbers is unimodal. (p_n does not depend on the probability distribution as long as the chance of getting the same number twice is 0.) Find p_1 to p_9 by simulation and guess a formula for p_n.

c) Find the exact value for p_n.

(14) *Palindromes.* We are given a list A of integers, with a length $n < 1000$.

a) Find a maximal subsequence $A[i, ..., j]$ such that $A[i] = A[j], A[i+1] = A[j-1], ...$, and print its length. The algorithm should have complexity $O(n^2)$.

b) Let $A[i]$ = random.randint(0, 9). Find by simulation the expected length of the longest palindrome.

(15) *Palindromes again.* We generate a sequence consisting of digits from the alphabet {0, 1, 2} each occurring with equal probability. Estimate the probability P that the sequence starts with a palindrome of length ≥ 2.

Hints: a) Look at sequences of length n. They will start with a palindrome with probability $p_n < P$, which we can estimate by simulation. For large n, $p_n \approx P$.

b) For small n you can find p_n by brute force. This is handy for testing whether your code is correct.

(16) *A Total Beetle Tour.* A beetle starts at a vertex of a cube and performs a random walk along its edges until it has visited all of the vertices. What is the expected time for such a Total Beetle Tour? (The beetle covers one edge in one time unit.)

(17) Suppose the random walk in Ex. 16 only stops when all the edges have been traversed. Find the expected time by simulation (you should get ≈ 48.5).

Additional Exercises for Chapters 1 to 4

(1) *A Two-Pile Game.* We have two piles containing a and b chips. Every second a pile X is chosen at random and a chip moved from X to the other pile Y. Let T be the waiting time until one pile is empty. Find by simulation a formula for the expected time $E(T) = f(a, b)$.

(2) *A Three-Pile Game.* We have three piles containing a, b, and c chips. Every second a pile X is chosen at random, then another pile Y is chosen at random, and a chip is moved from X to Y. Let T be the waiting time until one pile is empty. Write a simulation for calculating $E(T) = f(a, b, c)$. By choosing small values for a, b, and c, guess the $f(a, b, c)$ formula, which was first revealed in 1992 through computer experimentation.

(3) Write a program which computes $f(m, n) = \sqrt{m\sqrt{(m+1)...\sqrt{n}}}$. Explore $f(2, n)$ for ever larger values of n and study its convergence speed. Also, find $f(n, 2n)$, $f(n, 3n)$, $f(n, \infty)$ for large n.