The first lot fell to Jehoiarib, the second to Jedaiah, the third to Harim, the fourth to Seorim, the fifth to Malkijah, the sixth to Mijamin, the seventh to Hakkoz, the eighth to Abijah, the ninth to Jeshua, the tenth to Shekaniah, the eleventh to Eliashib, the twelfth to Jakim, the thirteenth to Huppah, the fourteenth to Jeshebeab, the fifteenth to Bilgah, the sixteenth to Immer, the seventeenth to Hezir, the eighteenth to Happizzez, the nineteenth to Pethahiah, the twentieth to Jehezkel, the twenty-first to Jakin, the twenty-second to Gamul, the twenty-third to Delaiah, and the twenty-fourth to Maaziah.

This was their appointed order of ministering when they entered the temple of the LORD, according to the regulations prescribed for them by their ancestor Aaron, as the LORD, the God of Israel, had commanded him.

— 1 Chronicles 24:7–19 (New International Version)

The ring worm is not ringed, nor is it worm. It is a fungus.
The puff adder is not a puff, nor can it add. It is a snake.
The funny bone is not funny, nor is it a bone. It is a nerve.
The fishstick is not a fish, nor is it a stick. It is a fungus.

— Matt Groening, "Life in Hell" (1986)

When Pierre-Simon Laplace described probability theory as "good sense reduced to a calculus," he intended to disparage neither good sense nor probability theory thereby.

— Lorraine Daston, "Enlightenment Calculations" (1994)

1 Discrete Probability

Before I start discussing randomized *algorithms* at all, I need to give a quick formal overview of the relatively small subset of probability theory that we will actually use. The first two sections of this note are deliberately written more as a review or reference than an introduction, although they do include a few illustrative (and hopefully helpful) examples.

1.1 Discrete Probability Spaces

A ***discrete***[1] ***probability space*** $(\Omega, \Pr)$ consists of a non-empty countable set Ω, called the ***sample space***, together with a ***probability mass function*** $\Pr\colon \Omega \to \mathbb{R}$ such that

$$\Pr[\omega] \ge 0 \text{ for all } \omega \in \Omega \qquad \text{and} \qquad \sum_{\omega \in \Omega} \Pr[\omega] = 1.$$

The latter condition implies that $\Pr[\omega] \le 1$ for all $\omega \in \Omega$. I don't know why the probability function is written with brackets instead of parentheses, but that's the standard;[2] just go with it. Here are a few simple examples:

- A fair coin: $\Omega = \{\mathsf{heads}, \mathsf{tails}\}$ and $\Pr[\mathsf{heads}] = \Pr[\mathsf{tails}] = 1/2$.
- A fair six-sided die: $\Omega = \{1,2,3,4,5,6\}$ and $\Pr[\omega] = 1/6$ for all $\omega \in \Omega$.
- A strangely loaded six-sided die: $\Omega = \{1,2,3,4,5,6\}$ with $\Pr[\omega] = \omega/21$ for all $\omega \in \Omega$. (For example, $\Pr[4] = 4/21$.)

[1] Correctly defining *continuous* (or otherwise uncountable) probability spaces and continuous random variables requires considerably more care and subtlety than the discrete definitions given here. There is *no* well-defined probability measure satisfying the discrete axioms when Ω is, for instance, an interval on the real line. This way lies the Banach-Tarski paradox.

[2] Or more honestly: one of many standards

- Bart's rock-paper-scissors strategy: $\Omega = \{\mathsf{rock}, \mathsf{paper}, \mathsf{scissors}\}$ and $\Pr[\mathsf{rock}] = 1$ and $\Pr[\mathsf{paper}] = \Pr[\mathsf{scissors}] = 0$.

Other common examples of countable sample spaces include the 52 cards in a standard deck, the 52! permutations of the cards in a standard deck, the natural numbers, the integers, the rationals, the set of all (finite) bit strings, the set of all (finite) rooted trees, the set of all (finite) graphs, and the set of all (finite) execution traces of an algorithm.

The precise choice of probability space is rarely important; we can usually implicitly define Ω to be the set of all possible tuples of values, one for each random variable under discussion.

1.1.1 Events and Probability

Subsets of Ω are usually called ***events***, and individual elements of Ω are usually called ***sample points*** or ***elementary events*** or ***atoms***. However, it is often useful to think of the elements of Ω as possible *states* of a system or *outcomes* of an experiment, and subsets of Ω as *conditions* that some states/outcomes satisfy and others don't.

The probability of an event A, denoted $\mathbf{Pr[A]}$, is defined as the sum of the probabilities of its constituent sample points:

$$\Pr[A] := \sum_{\omega \in A} \Pr[\omega]$$

In particular, we have $\Pr[\varnothing] = 0$ and $\Pr[\Omega] = 1$. Here we are extending (or overloading) the function $\Pr\colon \Omega \to [0,1]$ on atoms to a function $\Pr\colon 2^\Omega \to [0,1]$ on events.

For example, suppose we roll two fair dice, one red and the other blue. The underlying probability space consists of the sample space $\Omega = \{1,2,3,4,5,6\} \times \{1,2,3,4,5,6\}$ and the probabilities $\Pr[\omega] = 1/36$ for all $\omega \in \Omega$.

- The probability of rolling two 5s is $\Pr[\{(5,5)\}] = \Pr[(5,5)] = 1/36$.

- The probability of rolling a total of 6 is

$$\Pr\big[\{(1,5),(2,4),(3,3),(4,2),(5,1)\}\big] = \frac{5}{36}.$$

- The probability that the red die shows a 5 is

$$\Pr\big[\{(5,1),(5,2),(5,3),(5,4),(5,5),(5,6)\}\big] = \frac{1}{6}.$$

- The probability that at least one die shows a 5 is

$$\Pr\big[\{(1,5),(2,5),(3,5),(4,5),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(6,5)\}\big] = \frac{11}{36}.$$

- The probability that the red die shows a smaller number than the blue die is

$$\begin{aligned}\Pr\big[\{&(1,2),(1,3),(1,4),(1,5),(1,6),\\ &\qquad(2,3),(2,4),(2,5),(2,6),(3,4),\\ &\qquad\qquad(3,5),(3,6),(4,5),(4,6),(5,6)\}\big] = \frac{5}{12}.\end{aligned}$$

1.1.2 Combining Events

Because they are formally just sets, events can be combined using arbitrary set operations. However, in keeping with the intuition that events are *conditions*, these operations are usually written using Boolean logic notation $\wedge, \vee, \neg$ and vocabulary ("and, or, not") instead of the equivalent set notation $\cap, \cup, \overline{\cdot}$ and vocabulary ("intersection, union, complement"). For example, consider our earlier experiment rolling two fair six-sided dice, one red and the other blue.

$$\begin{aligned}
\Pr[\text{red } 5] &= 1/6 \\
\Pr[\text{two } 5\text{s}] &= \Pr[\text{red } 5 \wedge \text{blue } 5] = 1/36 \\
\Pr[\text{at least one } 5] &= \Pr[\text{red } 5 \vee \text{blue } 5] = 11/36 \\
\Pr[\text{at most one } 5] &= \Pr[\neg(\text{two } 5\text{s})] = 1 - \Pr[\text{two } 5\text{s}] = 35/36 \\
\Pr[\text{no } 5\text{s}] &= \Pr[\neg(\text{at least one } 5)] \\
&= 1 - \Pr[\text{at least one } 5] = 25/36 \\
\Pr[\text{exactly one } 5] &= \Pr[\text{at least one } 5 \wedge \text{at most one } 5] \\
&= \Pr[\text{red } 5 \oplus \text{blue } 5] = 5/18 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5] &= \Pr[\neg(\text{blue } 5) \vee \text{red } 5] = 31/36
\end{aligned}$$

(As usual, $p \Rightarrow q$ is just shorthand for $\neg p \vee q$; implication does *not* indicate causality!)

For any two events A and B with $\Pr[B] > 0$, the ***conditional probability of A given B*** is defined as

$$\mathbf{Pr}[\boldsymbol{A} \mid \boldsymbol{B}] := \frac{\Pr[A \wedge B]}{\Pr[B]}.$$

For example, in our earlier red-blue dice experiment:

$$\begin{aligned}
\Pr[\text{blue } 5 \mid \text{red } 5] &= \Pr[\text{two } 5\text{s} \mid \text{red } 5] = 1/6 \\
\Pr[\text{at most one } 5 \mid \text{red } 5] &= \Pr[\text{exactly one } 5 \mid \text{red } 5] \\
&= \Pr[\neg(\text{blue } 5) \mid \text{red } 5] = 5/6 \\
\Pr[\text{at least one } 5 \mid \text{at most one } 5] &= 2/7 \\
\Pr[\text{at most one } 5 \mid \text{at least one } 5] &= 10/11 \\
\Pr[\text{red } 5 \mid \text{at least one } 5] &= 6/11 \\
\Pr[\text{red } 5 \mid \text{at most one } 5] &= 1/7 \\
\Pr[\text{blue } 5 \mid \text{blue } 5 \Rightarrow \text{red } 5] &= 1/31 \\
\Pr[\text{red } 5 \mid \text{blue } 5 \Rightarrow \text{red } 5] &= 6/31 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5 \mid \text{blue } 5] &= 1/6 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5 \mid \text{red } 5] &= 1 \\
\Pr[\text{blue } 5 \Rightarrow \text{red } 5 \mid \text{red } 5 \Rightarrow \text{blue } 5] &= 26/31
\end{aligned}$$

Two events A and B are ***disjoint*** if they are disjoint as sets, meaning $A \cap B = \varnothing$. For example, in our two-dice experiment, the events "red 5" and "blue 5" and "total 5" are pairwise disjoint. Note that it is possible for $\Pr[A \wedge B] = 0$ even when the events A and B are not disjoint; consider the events "Bart plays paper" and "Bart does not play rock".

Two events A and B are ***independent*** if and only if $\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$. For example, in our two-dice experiment, the events "red 5" and "blue 5" are independent, but "red 5" and "total 5" are not.

More generally, a countable set of events $\{A_i \mid i \in I\}$ is ***fully*** or ***mutually independent*** if and only if $\Pr[\bigwedge_{i\in I} A_i] = \prod_{i\in I} A_i$. A set of events is ***k-wise independent*** if every subset of k events is fully independent, and ***pairwise independent*** if every pair of events in the set is independent. For example, in our two-dice experiment, the events "red 5" and "blue 5" and "total 7" are pairwise independent, but not mutually independent.

1.1.3 Identities and Inequalities

Fix n arbitrary events $A_1, A_2, \ldots, A_n$ from some sample space Ω. The following observations follow immediately from similar observations about sets.

- ***Union bound:*** For *any* events $A_1, A_2, \ldots A_n$, the definition of probability implies that
$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \;\le\; \sum_{i=1}^{n} \Pr[A_i].$$
The expression on the right counts each atom in the union of the events exactly once; the left summation counts each atom once for each event A_i that contains it.

- ***Disjoint union:*** If the events $A_1, A_2, \ldots A_n$ are pairwise disjoint, meaning $A_i \cap A_j = \varnothing$ for all $i \neq j$, the union bound becomes an equation:
$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \;=\; \sum_{i=1}^{n} \Pr[A_i].$$

- ***The principle of inclusion-exclusion*** describes a simple relationship between probabilities of unions (disjunctions) and intersections (conjunctions) of arbitrary events:
$$\Pr[A \vee B] \;=\; \Pr[A] + \Pr[B] - \Pr[A \wedge B]$$
This principle follows directly from elementary boolean algebra and the disjoint union bound:
$$\begin{aligned}
\Pr[A \vee B] + \Pr[A \wedge B] &= \Pr[(A \wedge B) \vee (A \wedge \overline{B}) \vee (\overline{A} \wedge B)] + \Pr[A \wedge B] \\
&= \big(\Pr[A \wedge B] + \Pr[A \wedge \overline{B}] + \Pr[\overline{A} \wedge B]\big) + \Pr[A \wedge B] \\
&= \big(\Pr[A \wedge B] + \Pr[A \wedge \overline{B}]\big) + \big(\Pr[\overline{A} \wedge B] + \Pr[A \wedge B]\big) \\
&= \Pr[A] + \Pr[B]
\end{aligned}$$
Inclusion-exclusion generalizes inductively any finite number of events as follows:
$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \;=\; 1 - \sum_{I \subseteq [1\,..\,n]} (-1)^{|I|} \Pr\left[\bigwedge_{i\in I} A_i\right]$$

- ***Independent union:*** For any pair A and B of independent events, we have
$$\begin{aligned}
\Pr[A \vee B] &= \Pr[A] + \Pr[B] - \Pr[A \wedge B] && \text{[inclusion-exclusion]} \\
&= \Pr[A] + \Pr[B] - \Pr[A]\Pr[B] && \text{[independence]} \\
&= 1 - (1 - \Pr[A])(1 - \Pr[B]).
\end{aligned}$$
More generally, if the events $A_1, A_2, \ldots A_n$ are *mutually* independent, then
$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] = 1 - \prod_{i=1}^{n} (1 - \Pr[A_i]).$$

- ***Bayes' Theorem:*** If events A and B both have non-zero probability, the definition of conditional probability immediately implies

$$\frac{\Pr[A \mid B]}{\Pr[A]} = \Pr[A \wedge B] = \frac{\Pr[B \mid A]}{\Pr[B]}.$$

and therefore

$$\Pr[A \mid B] \cdot \Pr[B] = \Pr[B \mid A] \cdot \Pr[A].$$

1.2 Random Variables

Formally, a ***random variable X*** is a function from a sample space Ω (with an associated probability measure) to some other *value set*. For example, the identity function on Ω is a random variable, as is the function that maps everything in Ω to Queen Elizabeth II, or any function mapping Ω to the real numbers. Random variables are almost universally denoted by upper-case letters.

A random variable is not random, nor is it a variable.

The value space of a random variable is commonly described either by an adjective preceding the phrase "random variable" or a noun replacing the word "variable". For example:

- A function from Ω to $\mathbb{Z}$ is called an *integer random variable* or a *random integer*.
- A function from Ω to $\mathbb{R}$ is called an *real random variable* or a *random real number*.
- A function from Ω to $\{0,1\}$ is called an *indicator random variable* or a *random bit*.

Since every integer is a real number, every integer random variable is also a real random variable; similarly, every random bit is also a random real number. Not all random variables are numerical; for example:

- A *random graph* is function from some sample space Ω to the set of all graphs.
- A *random point in the plane* is a function from some sample space Ω to $\mathbb{R}^2$.
- The function mapping every element of Ω to Queen Elizabeth II could be called a *random Queen of England*, or a *random monarch*, or (if certain unlikely conspiracy theories are taken seriously) a *random shape-shifting alien reptile*.

1.2.1 It is a fungus.

Although random variables are formally not variables at all, we typically describe and manipulate them *as if they were* variables representing unknown elements of their value sets, without referring to any particular sample space.

In particular, we can apply arbitrary functions to random variables by composition. Fir any random variable $X : \Omega \to V$ and any function $f : V \to V'$, the function $f(X) := f \circ X$ is a random variable over the value set V'. In particular, if X is a real random variable and α is any real number, then $X + \alpha$ and $\alpha \cdot X$ are also real random variables. More generally, if X and X' are random variables with value sets V and V', then for any function $f : V \times V' \to V''$, the function $f(X,X')$ is a random variable over V'', formally defined as

$$f(X,X')(\omega) := f(X(\omega), X'(\omega)).$$

These definitions extend in the obvious way to functions with an arbitrary number of arguments.

If ϕ is a boolean function or predicate over the value set of X, we implicitly identify the random variable $\phi(X)$ with the event $\{\omega \in \Omega \mid \phi(X(\omega))\}$. For example, if X is an integer random variable, then

$$\Pr[X = x] \;:=\; \Pr\big[\{\omega \mid X(\omega) = x\}\big]$$

and

$$\Pr[X \le x] \;:=\; \Pr\big[\{\omega \mid X(\omega) \le x\}\big]$$

and

$$\Pr[X \text{ is prime}] \;:=\; \Pr\big[\{\omega \mid X(\omega) \text{ is prime}\}\big].$$

In particular, we typically identify the boolean values **True** and **False** with the events Ω and $\varnothing$, respectively. Predicates with more than one random variable are handled similarly; for example,

$$\Pr[X = Y] \;:=\; \Pr\big[\{\omega \mid X(\omega) = Y(\omega)\}\big].$$

1.2.2 Expectation

For any real (or complex or vector) random variable X, the ***expectation of X*** is defined as

$$\mathbf{E}[\boldsymbol{X}] := \sum_x x \cdot \Pr[X = x].$$

This sum is always well-defined, because the set $\{x \mid \Pr[X = x] \ne 0\} \subseteq \Omega$ is countable. For *integer* random variables, the following definition is equivalent:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{x \ge 0} \Pr[X \ge x] - \sum_{x \le 0} \Pr[X \le x] \\ &= \sum_{x \ge 0} \big(\Pr[X \ge x] - \Pr[X \le -x]\big). \end{aligned}$$

If moreover A is an arbitrary event with non-zero probability, then the ***conditional expectation of X given A*** is defined as

$$\mathbf{E}[\boldsymbol{X} \mid \boldsymbol{A}] := \sum_x x \cdot \Pr[X = x \mid A] = \sum_x \frac{x \cdot \Pr[X = x \wedge A]}{\Pr[A]}$$

For any event A with $0 < \Pr[A] < 1$, we immediately have

$$\mathrm{E}[X] \;=\; \mathrm{E}[X \mid A] \cdot \Pr[A] \;+\; \mathrm{E}[X \mid \neg A] \cdot \Pr[\neg A].$$

In particular, for any random variables X and Y, we have

$$\mathrm{E}[X] = \sum_y \mathrm{E}[X \mid Y = y] \cdot \Pr[Y = y].$$

Two random variables X and Y are ***independent*** if, for all x and y, the events $X = x$ and $Y = y$ are independent. If X and Y are independent real random variables, then $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$. (However, this equation does *not* imply that X and Y are independent.) We can extend the notions of full, k-wise, and pairwise independence from events to random variables in similar fashion. In particular, if $X_1, X_2, \ldots, x_n$ are *fully independent* real random variables, then

$$\mathrm{E}\left[\prod_{i=1}^n X_i\right] = \prod_{i=1}^n \mathrm{E}[X_i].$$

Linearity of expectation refers to the following important fact: The expectation of any weighted sum of random variables is equal to the weighted sum of the expectations of those variables. More formally, for any real random variables $X_1, X_2, \ldots, X_n$ and any real coefficients $\alpha_1, \alpha_2, \ldots, \alpha_n$,

$$\mathrm{E}\left[\sum_{i=1}^{n}(\alpha_i \cdot X_i)\right] = \sum_{i=1}^{n}\big(\alpha_i \cdot \mathrm{E}[X_i]\big).$$

Linearity of expectation does *not* require the variables to be independent.

1.2.3 Examples

Consider once again our experiment with two standard fair six-sided dice, one red and the other blue. We define several random variables:

- R is the value (on the top face) of the red die.
- B is the value (on the top face) of the blue die.
- $S = R + B$ is the total value (on the top faces) of both dice.
- $\text{ꓤ} = 7 - R$ is the value on the *bottom* face of the red die.

The variables R and B are independent, as are the variables ꓤ and B, but no other pair of these variables is independent.

$$\begin{aligned}
\mathrm{E}[R] = \mathrm{E}[B] = \mathrm{E}[\text{ꓤ}] &= \frac{1+2+3+4+5+6}{6} = \frac{7}{2} \\
\mathrm{E}[R+B] &= \mathrm{E}[R] + \mathrm{E}[B] = 7 && \text{[linearity]} \\
\mathrm{E}[R+\text{ꓤ}] &= 7 && \text{[trivial distribution]} \\
\mathrm{E}[R+B+\text{ꓤ}] &= \mathrm{E}[R] + \mathrm{E}[B] + \mathrm{E}[\text{ꓤ}] = \frac{21}{2} && \text{[linearity]} \\
\mathrm{E}[R \cdot B] &= \mathrm{E}[R] \cdot \mathrm{E}[B] = \frac{49}{4} && \text{[independence]} \\
\mathrm{E}[R \cdot \text{ꓤ}] &= \frac{1\cdot 6 + 2 \cdot 5 + 3 \cdot 4}{3} = \frac{28}{3} \\
\mathrm{E}[R^2] &= \frac{1^2+2^2+3^2+4^2+5^2+6^2}{6} = \frac{91}{6} \\
\mathrm{E}[(R+B)^2] &= \mathrm{E}[R^2] + 2\,\mathrm{E}[RB] + \mathrm{E}[B^2] = \frac{329}{6} && \text{[linearity]} \\
\mathrm{E}[R+B \mid R=6] &= \mathrm{E}[R \mid R=6] + \mathrm{E}[B \mid R=6] && \text{[linearity]} \\
&= 6 + \mathrm{E}[B] = 19/2 && \text{[independence]} \\
\mathrm{E}[R \mid R+B=6] &= \frac{1+2+3+4+5}{5} = 3 \\
\mathrm{E}[R^2 \mid R+B=6] &= \frac{1^2+2^2+3^2+4^2+5^2}{5} = 11 \\
\mathrm{E}[R+B \mid R \cdot B=6] &= \frac{(1+6)+(2+3)}{2} = 6 \\
\mathrm{E}[R \cdot B \mid R+B=6] &= \frac{(1\cdot 5)+(2\cdot 4)+(3\cdot 3)+(4\cdot 2)+(5\cdot 1)}{5} = 7
\end{aligned}$$

1.3 Common Probability Distributions

A ***probability distribution*** assigns a probability to each possible value of a random variable. More formally, $X : \Omega \to V$ is a random variable over some probability space $(\Omega, \Pr)$, the probability distribution of X is the function $P : V \to [0,1]$ such that

$$P(x) = \Pr[X = x] = \sum\{\Pr(\omega) \mid X(\omega) = x\}.$$

The ***support*** of a probability distribution is the set of values with non-zero probability; this is a subset of the value set V. The following table summarizes several of the most useful discrete probability distributions.

name	intuition	parameters	support	$\Pr[X = x]$	$\mathrm{E}[X]$
trivial	Good ol' Rock, nothing beats that!	—	singleton set $\{a\}$	1	a
uniform	fair die roll	—	finite set $S \neq \varnothing$	$\dfrac{1}{\lvert S\rvert}$	$\dfrac{\sum S}{\lvert S\rvert}$
Bernoulli	biased coin flip	$0 \le p \le 1$	$\{0,1\}$	$\begin{cases} p & \text{if } x = 1 \\ 1-p & \text{if } x = 0 \end{cases}$	p
binomial	n biased coin flips	$0 \le p \le 1$ $n \ge 0$	$[0\,..\,n]$	$\binom{n}{x} p^x (1-p)^{n-x}$	np
geometric	#tails before first head	$0 < p \le 1$	$\mathbb{N}$	$(1-p)^x p$	$\dfrac{1-p}{p}$
negative binomial	#tails before nth head	$0 < p \le 1$ $n \ge 0$	$\mathbb{N}$	$\binom{n+x-1}{x}(1-p)^x p^n$	$\dfrac{n(1-p)}{p}$

Figure 1.1. Common discrete probability distributions.

- The ***trivial*** distribution describes the outcome of a "random" experiment that *always* has the same result. A trivially distributed random variable takes some fixed value with probability 1. Yes, this is still randomness.

- The ***uniform*** distribution assigns the same probability to every element of some finite non-empty set S. For example, if the random variable X is uniformly distributed over the integer range $[1\,..\,n]$, then $\Pr[X = x] = 1/n$ for each integer x between 1 and n, and $\mathrm{E}[X] = (n+1)/2$. This distribution models (idealized) fair coin flips, die rolls, and lotteries; consequently, this is what many people *incorrectly* think of as the definition of "random".

- The ***Bernoulli*** distribution models a random experiment (called a ***Bernoulli trial***) with two possible outcomes: ***success*** and ***failure***. The probability of success, usually denoted p, is a parameter of the distribution; the failure probability is often denoted $q = 1 - p$. Success and failure are usually represented by the *values* 1 and 0. Thus, every indicator random variable has a Bernoulli distribution, and its expected value is equal to its success probability:

$$\mathrm{E}[X] \;=\; \sum_x x \cdot \Pr[X = x] \;=\; 0 \cdot \Pr[X = 0] + 1 \cdot \Pr[X = 1] \;=\; \Pr[X = 1] \;=\; p.$$

  The special case $p = 1/2$ is a uniform distribution with two values: a fair coin flip. The special cases $p = 0$ and $p = 1$ are trivial distributions.

- The ***geometric*** distribution describes the number of independent Bernoulli trials (all with the same success probability p) before the first success. If X is a geometrically distributed random variable, then $X = x$ if and only if the first x Bernoulli trials fail and the $(x+1)$th trial succeeds:
$$\Pr[X = x] \;=\; \left(\prod_{i=1}^{x} \Pr[i\text{th trial fails}]\right) \cdot \Pr[(x+1)\text{th trial succeeds}] \;=\; (1-p)^x p.$$

- The ***binomial*** distribution is the sum of n independent Bernoulli distributions, all with the same probability p of success. If X is a binomially distributed random variable, then $X = x$ if and only if x of the n trials succeed and $n - x$ fail:
$$\Pr[X = x] = \binom{n}{x} p^x (1-p)^{n-x}.$$
If $n = 1$, this is just the Bernoulli distribution.

- The ***negative binomial*** distribution describes the number of independent Bernoulli trials (all with the same success probability p) that fail before the nth successful trial. If X is a negative-binomially distributed random variable, then $X = x$ if and only if exactly x of the first $n + x - 1$ trials are failures, and the $(n + x)$th trial is a success:
$$\Pr[X = x] = \binom{n+x-1}{x}(1-p)^x p^n$$
If $n = 1$, this is just the geometric distribution.

1.4 Coin Flips

Suppose you are given a coin and you are asked generate a uniform random bit. We distinguish between two types of coins: A coin is *fair* if the probability of heads (1) and the probability of tails (0) are both exactly 1/2. A coin where one side is more likely than the other is said to be *biased*. Actual physical coins are reasonable approximations of abstract fair coins for most purposes, at least if they're flipped high into the air and allowed to bounce.[3] Physical coins can be biased by bending them.

1.4.1 Removing Unknown Bias

In 1951, John von Neumann discovered the following simple technique to simulate fair coin flips using an arbitrarily biased coin, *even without knowing the bias*. Flip the biased coin twice. If the two flips yield different results, return the first result; otherwise, repeat the experiment from scratch.

```
VonNeumannCoin():
  x ← BiasedCoin()
  y ← BiasedCoin()
  if x ≠ y
      return x
  else
      return VonNeumannCoin()
```

[3]Persi Diaconis, Susan Holmes, and Richard Montgomery published a thorough analysis of physical coin-flipping in 2007, which concluded (among other things) that coins that are flipped vigorously and then caught come up in the same state they started about 51% of the time. The small amount of bias arises because flipping coins tend to precess as they rotate. Letting the coin bounce instead of catching them appears to remove the bias from precession.

This is weird sort of algorithm, isn't it? There is *no* upper bound on the worst-case running time; in principle, the algorithm could run *forever*, because the biased coin always just happens to flip heads. Nevertheless, I claim that this is a useful algorithm for generating fair random bits. We need two technical assumptions:

(1) The biased coin always flips heads with the same fixed (but unknown!) probability p. To simplify notation, we let $q = 1 - p$ denote the probability of flipping tails. For example, a fair coin would have $p = q = 1/2$.

(2) All flips of the biased coin are mutually independent.

First, I claim that *if* the algorithm halts, then it returns a uniformly distributed random bit. Because the two biased coin flips are independent, we have

$$\Pr[x = 0 \wedge y = 1] = \Pr[x = 1 \wedge y = 0] = pq,$$

and therefore (assuming $pq > 0$)

$$\Pr[x = 0 \wedge y = 1 \mid x \neq y] = \Pr[x = 1 \wedge y = 0 \mid x \neq y] = \frac{pq}{2pq} = \frac{1}{2}.$$

Because the biased coin flips are mutually independent, the same analysis applies without modification to every recursive call to VonNeumannCoin. Thus, if *any* recursive call returns a bit, that bit is uniformly distributed.

Now let T denote the actual running time of this algorithm; T is a random variable that depends on the biased coin flips.[4] We can compute the expected running time $\mathrm{E}[T]$ by considering the conditional expectations in two cases: The first two flips are either different or equal.

$$\mathrm{E}[T] \;=\; \mathrm{E}[T \mid x \neq y] \cdot \Pr[x \neq y] \;+\; \mathrm{E}[T \mid x = y] \cdot \Pr[x = y]$$

Because the two biased coin flips are independent, we have

$$\Pr[x \neq y] = \Pr[x = 0 \wedge y = 1] + \Pr[x = 1 \wedge y = 0] = 2pq$$

and therefore $\Pr[x = y] = 1 - 2pq$. If the first two coin flips are different, the algorithm ends after two flips; thus, $\mathrm{E}[T \mid x \neq y] = 2$. Finally, if the first two coin flips are the same, the experiment starts over from scratch after the first two flips, so $\mathrm{E}[T \mid x = y] = 2 + \mathrm{E}[T]$. Putting all the pieces together, we have

$$\mathrm{E}[T] = 2 \cdot 2pq + (2 + \mathrm{E}[T]) \cdot (1 - 2pq).$$

Solving this equation for $\mathrm{E}[T]$ yields the solution $\mathbf{E[T] = 1/pq}$. For example, if $p = 1/3$, the expected number of coin flips is $9/2$.

Alternatively, we can think of VonNeumannCoin as performing an experiment with a different biased coin, which returns a result ("heads") with probability $2pq$. Thus, the expected number of *unsuccessful* iterations ("tails" before the first "head") is a geometric random variable with expectation $1/2pq - 1$, and thus the expected number of iterations is $1/2pq$. Because each iteration flips two coins, the expected number of coin flips is $1/pq$.

[4]Normally we use $T(n)$ to denote the worst-case running time of an algorithm as a function of some input parameter n, but this algorithm has no input parameters!

1.4.2 Removing Known Bias

But what if we *know* that $p = 1/3$? In that case, the following algorithm simulates a fair coin with fewer biased flips (on average):

```
FAIRCOIN():
  x ← BIASEDCOIN(1/3)
  y ← BIASEDCOIN(1/3)
  if x ≠ y              ⟨⟨probability 4/9⟩⟩
      return 0
  else if x = y = 1     ⟨⟨probability 4/9⟩⟩
      return 1
  else                  ⟨⟨probability 1/9⟩⟩
      return FAIRCOIN()
```

The algorithm returns a fair coin because

$$\Pr[x \neq y] = \frac{4}{9} = \Pr[x = y = 0].$$

The expected number of flips satisfies the equation

$$\mathrm{E}[T] = 2 + \frac{1}{9}\mathrm{E}[T],$$

which implies that $\mathrm{E}[T] = 9/4$, a factor of 2 better than von Neumann's algorithm.

1.5 Pokémon Collecting

A distressingly large fraction of my daughters' friends are obsessed with Pokémon—not the cartoon or the mobile game, but the collectible card game. The Pokémon Company sells small packets, each containing half a dozen cards, each describing a different Pokémon character. The cards can be used to play a complex turn-based combat game; the more cards a player owns, the more likely they are to win. So players are strongly motivated to collect as many cards, and in particular, as many *different* cards, as possible. Pokémon reinforces this message with their oh-so-subtle theme song "Gotta Catch 'Em All!" Unfortunately, the packets are opaque; the only way to find out which cards are in a pack is to buy the pack and tear it open.[5]

Let's consider the following oversimplified model of the Pokémon-collection process. In each trial, we purchase one Pokémon card, chosen independently and uniformly at random from the set of n possible card types. We repeat these trials until we have purchased at least one of each type of card. This problem was first considered by the French mathematician Abraham de Moivre in his seminal 1712 treatise *De Necessitate ut Caperent Omnium Eorum*.[6]

1.5.1 After n Trials

How many different types of Pokémon do we actually own after we buy n cards? Obviously in the worst case, we might just have n copies of one Pokémon,[7] but that's not particularly likely. To

[5] See also: cigarette cards, Dixie cups, baseball cards, Pez dispensers, Beanie Babies, Iwako puzzle erasers, Shopkins, and Guys Under Your Supervision.

[6] The actual title was *De Mensura Sortis seu; de Probabilitate Eventuum in Ludis a Casu Fortuito Pendentibus*, which means "On the measurement of chance, or on the probability of events in games depending on fortuitous chance".

[7] "Dave Guy!"

analyze the *expected* number of types that we own, we introduce an incredibly useful technique that lets us exploit linearity of expectation: decomposing more complex random variables into ***sums of indicator variables***.

For each index i, define an indicator variable $X_i = [\text{we own Pokémon } i]$ so that $X = \sum_i X_i$ is the number of cards we own. Linearity of expectation implies

$$\mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \Pr[X_i = 1].$$

The probability that we *don't* own card i is $(1-1/n)^n \approx 1/e$, so

$$\mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] \approx \sum_{i=1}^{n} (1-1/e) = (1-1/e)n \approx 0.63212n.$$

In other words, after buying n cards, we expect to own a bit less than 2/3 of the Pokémon.

Similar calculations implies that we expect to own about 86% of the Pokémon after buying $2n$ cards, about 95% after buying $3n$ cards, about 98% after buying $4n$ cards, and so on.

1.5.2 Gotta Catch 'em *All*

So how many Pokémon packs do we need to buy to catch 'em *all*? Obviously in the worst case, we might *never* have a complete collection[8], but assuming each type of card has some non-zero probability of being in each pack, the *expected* number of packs we need to buy is finite. Let $T(n)$ denote the number of packs (or "time") required to collect all n Pokémon. For purposes of analysis, we partition the random purchasing algorithm into n *phases*, where the ith phase ends just after we see the ith distinct card type. Let us write

$$T(n) = \sum_i T_i(n)$$

where $T_i(n)$ is the number of cards bought during the ith phase. Linearity of expectation implies

$$\mathrm{E}[T(n)] = \sum_i \mathrm{E}[T_i(n)].$$

We can think of each card purchase as a biased coin flip, where "heads" means "got a new Pokémon" and "tails" means "got a Pokémon we already own". For each index i, the probability of heads (that is, the probability of a single purchase being a new Pokémon) is exactly $p = (n-i+1)/n$: each of the n Pokémon is equally likely, and there are $n-i+1$ Pokémon that we don't already own. By our earlier analysis, the expected number of flips until the first head is $\mathrm{E}[T_i] = 1/p = n/(n-i+1)$. We conclude that

$$\mathrm{E}[T(n)] \;=\; \sum_i \mathrm{E}[T_i(n)] \;=\; \sum_{i=1}^{n} \frac{n}{n-i+1} \;=\; \sum_{j=1}^{n} \frac{n}{j} \;=\; nH_n.$$

Here $\boldsymbol{H_n}$ denotes the nth ***harmonic number***, defined recursively as

$$H_n = \begin{cases} 0 & \text{if } n = 0 \\ H_{n-1} + \dfrac{1}{n} & \text{otherwise} \end{cases}$$

[8]"Dave Guy!"

Approximating the summation $H_n = \sum_{i=1}^{n} \frac{1}{i}$ above and below by integrals implies the bounds

$$\ln(n+1) \le H_n \le (\ln n) + 1.$$

Thus, the expected number of cards we need to buy to get all n Pokémon is $\Theta(n \log n)$.

In particular, to catch all 150 of the original Pokémon, we should expect to buy $150 \cdot H_{150} \approx 838.67709$ cards, and to own at least one copy of each of the 9184 Pokémon card types available in 2013, we should expect to buy $9184 \cdot H_{9184} \approx 89107.65186$ cards. (In practice, of course, this estimate is far too low, because some cards are considerably more common than others.)

1.6 Random Permutations

Now suppose we are given a deck of n (Pokémon?) cards and are asked to shuffle them. Ideally, we would like an algorithm that produces each of the $n!$ possible permutations of the deck with the same probability $1/n!$.

There are many such algorithms, but the gold standard is ultimately based on the millennia-old tradition of drawing or casting lots. "Lots" are traditionally small pieces of wood, stone, or paper, which were blindly drawn from an opaque container. The following algorithm takes a set L of n distinct Lots (arbitrary objects) as input and returns an array $R[1..n]$ containing a Random permutation of those n lots.

```
DrawLots(L):
  n ← |L|
  for i ← 1 to n
      remove a random lot x from L
      R[i] ← x
  return R[1..n]
```

There are exactly $n!$ possible outcomes for this algorithm—exactly n choices for the first lot, then exactly $n-1$ choices for the second lot, and so on—each with exactly the same probability and each leading to a different permutation. Thus, every permutation of lots is equally likely to be output.

A modern formulation of lot-casting was described by (and is frequently misattributed to) statisticians Ronald Fisher and Frank Yates in 1938. Fisher and Yates formulated their algorithm as a method for randomly reordering a List of numbers. In their original formulation, the algorithm repeatedly chooses at random a number from the input list that has not been previously chosen, adds the chosen number to the output list, and then strikes the chosen number from the input list.

We can implement this formulation of the algorithm using a secondary boolean array $Chosen[1..n]$ indicating which items have already been chosen. The randomness is provided by a subroutine ***Random(n)***, which returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, n\}$ in $O(1)$ time; in other words, Random(n) simulates a fair n-sided die.

```
FisherYates(L[1..n]):
    for i ← 1 down to n
        Chosen[i] ← False
    for i ← n down to 1
        repeat
            r ← Random(n)
        until ¬Chosen[r]
        R[i] ← L[r]
        Chosen[r] ← True
    return R[1..n]
```

The repeat-until loop chooses an index r uniformly at random from the set of previously unchosen indices. Thus, this algorithm really is an implementation of DrawLots. But now choosing the next random ~~lot~~ element may require several iterations of the repeat-until loop. How slow is this algorithm?

In fact, FisherYates is equivalent to our earlier Pokémon-collecting algorithm! Each call to Random is a purchase, and $Chosen[i]$ indicates whether we've already purchased the ith Pokémon. By our earlier analysis, the expected number of calls to Random before the algorithm halts is *exactly* nH_n. We conclude that this algorithm runs in $\Theta(n \log n)$ ***expected time***.[9]

A most efficient implementation of lot-casting, which permutes the input array in place, was described by Richard Durstenfeld in 1961. In full accordance with Stigler's Law, this algorithm is almost universally called "the Fisher-Yates shuffle".[10]

```
SelectionShuffle(A[1..n]):
    for i ← n down to 1
        swap A[i] ↔ A[Random(i)]
```

The algorithm clearly runs in $O(n)$ time. Correctness follows from exactly the same argument as DrawLots: There are $n!$ equally likely possibilities from the n calls to Random—n for the first call, $n-1$ for the second, and so on—each leading to a different output permutation.

Although it may not appear so at first glance, SelectionShuffle is an implementation of lot-casting. After each iteration of the main loop, the suffix $A[i\,..\,n]$ (on the *R*ight) plays the role of R, storing the previously chosen input elements, and the prefix $A[1\,..\,i-1]$ (on the *L*eft) plays the role of L, containing all the unchosen input elements. Unlike the original FisherYates algorithm, SelectionShuffle changes the *order* of the unchosen elements as it runs. Fortunately, and obviously, that order is utterly irrelevant; only the *set* of unchosen elements matters.

We can also uniformly shuffle by reversing the order of the loop in the previous algorithm. Again, this algorithm is usually misattributed to Fisher and Yates.

```
InsertionShuffle(A[1..n]):
    for i ← 1 to n
        swap A[i] ↔ A[Random(i)]
```

Again, correctness follows from the observation that there are $n!$ equally likely output permutations. Alternatively, we can argue inductively that for every index i, the prefix $A[1\,..\,i]$ is

[9]In later editions of Fisher and Yates' monograph, they instead described a different algorithm due to C. Radhakrishna Rao, dismissing their earlier method as "tiresome, since each [item] must be deleted from a list as it is selected and a fresh count made for each further selection."

[10]However, some authors call this algorithm the "Knuth shuffle", because Donald Knuth described it in his landmark *Art of Computer Programming*, even though he attributed the algorithm to Durstenfeld in the first edition, and to Fisher and Yates in the second. It's actually rather shocking that Knuth did not attribute the algorithm to Aaron, citing the First Chronicles.

uniformly shuffled after the ith iteration of the loop. Alternatively, we can observe that running InsertionShuffle is the same as running SelectionShuffle *backward in time*—essentially putting the lots back into the bag—and that the inverse of a uniformly-distributed permutation is also uniformly distributed.

(The names SelectionShuffle and InsertionShuffle for these two variants are non-standard. SelectionShuffle randomly *selects* the next card and then adds to one end of the random permutation, just as selection sort repeatedly selects the largest element of the unsorted portion of the array. InsertionShuffle randomly *inserts* the first card in the untouched portion of the array into the random permutation, just as insertion sort repeatedly inserts the next item in the unsorted portion of the input into the sorted portion.)

1.7 Properties of Random Permutations

★★★

- For any subsequence of indices: the set of values and the permutation of those values are uniformly and independently distributed.
- For any subsequence of values: the set of indices and the permutation of those indices are uniformly and independently distributed.
- For example: In a randomly shuffled deck, the expected number of hearts among the first 5 face cards is 5/4.

Exercises

Several of these problems refer to decks of playing cards. A standard (Anglo-American) deck of 52 playing cards contains 13 cards in each of four suits: ♠ (spades), ♥ (hearts), ♦ (diamonds), and ♣ (clubs). The 13 cards in each suit have distinct *ranks*: A (ace), 2 (deuce), 3 (trey), 4, 5, 6, 7, 8, 9, 10, J (jack), Q (queen), and K (king). Cards are normally named by writing their rank followed by their suit; for example, J♠ is the jack of spades, and 10♥ is the ten of hearts. For purposes of comparing ranks in the problems below, aces have rank 1, jacks have rank 11, queens have rank 12, and kings have rank 13; for example, J♠ has higher rank than 8♦, but lower rank than Q♣.

---

1. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

   Rosencrantz: **H H** T
   Guildenstern: H T **H H**

   Rosencrantz: T
   Guildenstern: (no flips)

   Rosencrantz: **H H H** T
   Guildenstern: T H H T H H T H T T **H H H**

   (a) What is the expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz happens to flip k heads in a row on his turn. What is the expected number of flips in Guildenstern's next turn?

(c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

Prove that your answers are correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! Full credit requires *exact* answers, but a correct asymptotic bound (as a function of k) in part (b) is worth significant partial credit.

2. After sending his loyal friends Rosencrantz and Guildenstern off to Norway, Hamlet decides to amuse himself by repeatedly flipping a fair coin until the sequence of flips satisfies some condition. For each of the following conditions, compute the *exact* expected number of flips until that condition is met. Some conditions depend on a positive integer parameter n.

   (a) Hamlet flips n times.

   (b) Hamlet flips heads.

   (c) Hamlet flips both heads and tails (in different flips, of course).

   (d) Hamlet flips heads twice.

   (e) Hamlet flips heads twice in a row.

   (f) Hamlet flips heads followed immediately by tails.

   (g) Hamlet flips the sequence heads, tails, heads, tails.

   (h) Hamlet flips heads n times.

   (i) Hamlet flips heads n times in a row.

   (j) Hamlet flips more heads than tails.

   (k) Hamlet flips the same positive number of heads and tails.

   (l) Either Hamlet flips more heads than tails or he flips n times, whichever comes first.

   For each condition, prove that your answer is correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! Correct asymptotic bounds for the conditions that depend on n are worth significant partial credit.

3. Suppose you have access to a function FairCoin that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. Consider the following randomized algorithm for generating biased random bits.

```
OneInThree:
  if FairCoin = 0
      return 0
  else
      return 1 − OneInThree
```

   (a) Prove that OneInThree returns 1 with probability 1/3.

   (b) What is the *exact* expected number of times that this algorithm calls FairCoin?

   (c) Now suppose instead of FairCoin you are given a subroutine BiasedCoin that returns an independent random bit equal to 1 with some fixed *but unknown* probability p, in $O(1)$ time. Describe an algorithm OneInThree that returns either 0 or 1 with equal probability, using BiasedCoin as its only source of randomness.

(d) What is the *exact* expected number of times that your OneInThree algorithm calls BiasedCoin?

4. Describe an algorithm that takes a real number $0 \le p \le 1$ as input, and returns an independent random bit that is equal to 1 with the given probability p, using independent fair coin flips as the only source of randomness. What is the expected running time of your algorithm (as a function of p)?

5. (a) Describe an algorithm that simulates a fair three-sided die, using independent fair coin flips as the only source of randomness. Your algorithm should return 1, 2, or 3, each with probability 1/3. What is the expected number of coin flips used by your algorithm?

   (b) Describe an algorithm that simulates a fair coin flip, using independent rolls of a fair three-sided die as the only source of randomness. What is the expected number of die rolls used by your algorithm?

   *(c) Describe an algorithm that simulates a fair n-sided die, using independent rolls of a fair k-sided die as the only source of randomness. What is the expected number of die rolls used by your algorithm (as a function of n and k)? You may assume n and k are relatively prime.

6. Describe an algorithm FairDie that returns an integer chosen uniformly at random from the set $\{1, 2, 3, 4, 5, 6\}$, using an algorithm LoadedDie that returns an algorithm from the same set with some fixed *but unknown* non-trivial probability distribution. What is the expected number of times your FairDie algorithm calls LoadedDie? *[Hint:* $3! = 6$.*]*

7. (a) Suppose you have access to a function FairCoin that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. Describe and analyze an algorithm Random(n) that returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$, given a non-negative integer n as input, using FairCoin as its only source of randomness.

   (b) Suppose you have access to a function FairCoins(k) that returns an integer chosen uniformly and independently at random from the set $\{0, 1, \ldots, 2^k - 1\}$ in $O(1)$ time, given *any* non-negative integer k as input. Describe and analyze an algorithm Random(n) that returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$, given *any* non-negative integer n as input, using FairCoins as its only source of randomness.

8. You are applying to participate in this year's Trial of the Pyx, the annual ceremony where samples of all British coinage are tested, to ensure that they conform as strictly as possible to legal standards. As a test of your qualifications, your interviewer at the Worshipful Company of Goldsmiths has given you a bag of n commemorative Alan Turing half-guinea coins, exactly two of which are counterfeit. One counterfeit coin is very slightly lighter than a genuine Turing; the other is very slightly heavier. Together, the two counterfeit coins have *exactly* the same weight as two genuine coins. Your task is to identify the two counterfeit coins.

The weight difference between the real and fake coins is too small to be detected by anything other than the Royal Pyx Coin Balance. You can place any two disjoint sets of coins in each of the Balance's two pans; the Balance will then indicate which of the two subsets has larger total weight, or that the two subsets have the same total weight. Unfortunately, each use of the Balance requires completing a complicated authorization form (in triplicate), submitting a blood sample, and scheduling the Royal Bugle Corps, so you *really* want to use the Balance as few times as possible.

(a) Suppose you *randomly* choose $n/2$ of your n coins to put on one pan of the Balance, and put the remaining $n/2$ coins on the other pan. What is the probability that the two subsets have equal weight?

(b) Describe and analyze a randomized algorithm to identify the two fake coins. What is the expected number of times your algorithm uses the Balance?

9. Consider the following algorithm for shuffling a deck of n cards, initially numbered in order from 1 on the top to n on the bottom. At each step, we remove the top card from the deck an insert it randomly back into in the deck, choosing one of the n possible positions uniformly at random. The algorithm ends immediately after we pick up card $n-1$ and insert it randomly into the deck.

   (a) Prove that this algorithm uniformly shuffles the deck, so that each permutation of the deck has equal probability. *[Hint: Prove that at all times, the cards below card $n-1$ are uniformly shuffled.]*

   (b) Prove that before the algorithm ends, the deck is *not* uniformly shuffled.

   (c) What is the expected number of steps before this algorithm ends?

10. Suppose we want to write an efficient algorithm Shuffle($A[1..n]$) that randomly permutes the input array, so that each of the $n!$ permutations is equally likely.

   (a) Prove that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

```
NaiveShuffle(A[1..n]):
  for i ← 1 to n
      swap A[i] ↔ A[Random(n)]
```

   (The only difference from InsertionShuffle is that the argument to Random is n instead of i.)

   (b) The algorithm BufferShuffle, shown in Figure 1.2, takes an extra parameter b describing the size of the internal buffer array $B[1..b]$.

      i. Prove that BufferShuffle is correct for all $b \geq n$.

      ii. What is the expected running time of BufferShuffle when $b = n$?

      iii. What is the expected running time of BufferShuffle when $b = 2n$?

   *(c) Prove that the algorithm RaoShuffle, shown in Figure 1.2, is correct and analyze its expected running time. This is the algorithm of C. Radhakrishna Rao that Fisher and Yates found less "tiresome" than their own.

```
BufferShuffle(n, b):
  ⟨⟨Initialize buffer⟩⟩
  for j ← 1 to 2n
      B[j] ← Null
  ⟨⟨Copy items randomly into buffer⟩⟩
  for i ← 1 to n
      j ← Random(b)
      while (B[j] == Null)
          j ← Random(n)
      B[j] ← A[i]
  ⟨⟨Compress buffer into input array⟩⟩
  i ← 1
  for j ← 1 to b
      if B[j] ≠ Null
          A[i] ← B[j]
          i ← i + 1
```

```
RaoShuffle(A[1..n]):
  ⟨⟨Initialize n lists⟩⟩
  for j ← 1 to n
      ℓ[j] ← 0
  ⟨⟨Add each A[i] to a random list⟩⟩
  for i ← 1 to n
      ℓ[j] ← ℓ[j] + 1
      L[j][ℓ[j]] ← A[i]
  ⟨⟨Recursively shuffle the lists⟩⟩
  for j ← 1 to n
      if ℓ[j] > 1
          Shuffle(L[j][1..ℓ[j]])
  ⟨⟨Concatenate the lists⟩⟩
  i ← 0
  for j ← 1 to n
      for k ← 1 to ℓ[j]
          A[i] ← L[j][k]
          i ← i + 1
```

Figure 1.2. Left: Shuffling using a buffer array. Right: Rao's less "tiresome" shuffling algorithm.

11. (a) Prove that the following algorithm, modeled after quicksort, uniformly permutes its input array, meaning each of the $n!$ possible output permutations is equally likely. *[Hint: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.]*

```
QuickShuffle(A[1..n]):
  if n ≤ 1
      return
  j ← 1;  k ← n
  while j ≤ k
      with probability 1/2
          swap A[j] ↔ A[k]
          k ← k − 1
      else
          j ← j + 1
  QuickShuffle(A[1..k])
  QuickShuffle(A[j..n])
```

(b) Prove that QuickShuffle runs in $O(n \log n)$ expected time. *[Hint: This will be much easier after reading the next chapter.]*

(c) Prove that the algorithm MergeShuffle shown on the next page, which is modeled after mergesort, does ***not*** uniformly permute its input array.

(d) Describe how to modify RandomMerge so that MergeShuffle ***does*** uniformly permute its input array, and prove that your modification is correct. *[Hint: Modify the line in red.]*

```
MergeShuffle(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MergeShuffle(A[1..m])
      MergeShuffle(A[m+1..n])
      RandomMerge(A[1..n], m)
```

```
RandomMerge(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else with probability 1/2
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

12. Clock Solitaire is played with a standard deck of playing cards. To set up the game, deal the cards face down into 13 piles of four cards each, one in each of the "hour" positions of a clock and one in the center. Each pile corresponds to a particular rank—A through Q in clockwise order for the hour positions, and K for the center. To start the game, turn over a card in the center pile. Then repeatedly turn over a card in the pile corresponding to the value of the previous card. The game ends when you try to turn over a card from a pile whose four cards are already face up. (This is always the center pile—why?) You win if and only if every card is face up when the game ends.

    What is the *exact* probability that you win a game of Clock Solitaire, assuming that the cards are permuted uniformly at random before they are dealt into their piles?

13. Professor Jay is about to perform a public demonstration with two decks of cards, one with red backs ("the red deck") and one with blue backs ("the blue deck"). Both decks lie face-down on a table in front of the good Professor, shuffled so that every permutation of each deck is equally likely.

    To begin the demonstration, Professor Jay turns over the top card from each deck. If one of these two cards is the three of clubs (3♣), the demonstration ends immediately. Otherwise, the good Professor repeatedly hurls the cards he just turned over into the thick, pachydermatous outer melon layer of a nearby watermelon, and then turns over the next card from the top of each deck. The demonstration ends the first time a 3♣ is turned over. Thus, if 3♣ is the last card in both decks, the demonstration ends with 102 cards embedded in the watermelon, that most prodigious of household fruits.

    (a) What is the *exact* expected number of cards that Professor Jay hurls into the watermelon?

    (b) For each of the statements below, give the *exact* probability that the statement is true of the ***first*** pair of cards Professor Jay turns over.

      i. Both cards are threes.
      ii. One card is a three, and the other card is a club.
      iii. If (at least) one card is a heart, then (at least) one card is a diamond.
      iv. The card from the red deck has higher rank than the card from the blue deck.

    (c) For each of the statements below, give the *exact* probability that the statement is true of the ***last*** pair of cards Professor Jay turns over.

i. Both cards are threes.
ii. One card is a three, and the other card is a club.
iii. If (at least) one card is a heart, then (at least) one card is a diamond.
iv. The card from the red deck has higher rank than the card from the blue deck.

14. Penn and Teller agree to play the following game. Penn shuffles a standard deck of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

    The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[11] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

    (a) What is the expected number of cards that Teller draws?
    (b) What is the expected *maximum* value among the cards Teller gives to Penn?
    (c) What is the expected *minimum* value among the cards Teller gives to Penn?
    (d) What is the expected number of cards that Teller gives to Penn? *[Hint: Let* $13 = n$*.]*

15. Consider a random walk on a path with vertices numbered $1, 2, \dots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n.

    (a) Prove that the probability that the walk ends by falling off the *right* end of the path is exactly $1/(n+1)$.
    (b) Prove that if we start at vertex k, the probability that we fall off the *right* end of the path is exactly $k/(n+1)$.
    (c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n.
    (d) What is the *exact* expected length of the random walk if we start at vertex k, as a function of n and k? Prove your result is correct. (For partial credit, give a tight Θ-bound for the case $k = (n+1)/2$, assuming n is odd.)

    [Hint: Trust the recursion fairy.]

16. Suppose n lights labeled $0, \dots, n-1$ are placed clockwise around a circle. Initially, every light is off. Consider the following random process.

[11] Specifically, he hurls it directly into the back of Penn's right hand.

```
LIGHTTHECIRCLE(n):
  k ← 0
  turn on light 0
  while at least one light is off
      with probability 1/2
          k ← (k + 1) mod n
      else
          k ← (k − 1) mod n
      if light k is off, turn it on
```

(a) Let $p(i,n)$ denote the probability that the last light turned on by LightTheCircle$(n,0)$ is light i. For example, $p(0,2) = 0$ and $p(1,2) = 1$. Find an exact closed-form expression for $p(i,n)$ in terms of n and i. Prove your answer is correct.

(b) Give the tightest upper bound you can on the expected running time of this algorithm. *[Hint: You may find the previous exercise helpful.]*

17. Let T be an arbitrary, ***not*** necessarily balanced, binary tree T with n nodes.

(a) Consider a random walk downward from the root of T, as described by the following algorithm.

```
RANDOMTREEDESCENT(T):
  v ← T.root
  while v ≠ NULL
      with probability 1/2
          v ← v.left
      else
          v ← v.right
```

Find a tight bound on the worst-case expected running time of this algorithm, as a function of n, and prove your answer is correct. What is the worst-case tree?

(b) Now consider a different random walk starting at the root of T, as described by the following algorithm.

```
RANDOMTREEWALK(T):
  v ← T.root
  while v ≠ NULL
      with probability 1/3
          v ← v.left
      else with probability 1/3
          v ← v.right
      else
          v ← v.parent
```

Find a tight bound on the worst-case expected running time of this algorithm, as a function of n, and prove your answer is correct. What is the worst-case tree?