16.3 Using Table Lookup to Improve Performance

You might not want to try running `crapsSim(100000000, 10)` at home. It takes a long time to complete on most computers. That raises the question of whether there is a simple way to speed up the simulation.

The complexity of `crapsSim` is O(playHand)*handsPerGame*numGames. The running time of `playHand` depends upon the number of times the loop in it is executed. In principle, the loop could be executed an unbounded number of times since there is no bound on how long it could take to roll either a 7 or the point. In practice, of course, we have every reason to believe it will always terminate.

Notice, however, that the result of a call to `playHand` does not depend on how many times the loop is executed, but only on which exit condition is reached. For each possible point, one can easily calculate the probability of rolling that point before rolling a 7. For example, using a pair of dice one can roll a 4 in three different ways: <1, 3>, <3, 1>, and <2, 2>; and one can roll a 7 in six different ways: <1, 6>, <6, 1>, <2, 5>, <5, 2>, <3, 4>, and <4, 3>. Therefore, exiting the loop by rolling a 7 is twice as likely as exiting the loop by rolling a 4.

Figure 16.4 contains an implementation of `playHand` that exploits this thinking. We first compute the probability of making the point before rolling a 7 for each possible value of the point, and store those values in a dictionary. Suppose, for example, that the point is 8. The shooter continues to roll until he either rolls the point or rolls craps. There are five ways of rolling an 8 (<6,2>, <2,6>, <5,3>, <3,5>, and <4,4>) and six ways of rolling a 7. So, the value for the dictionary key `8` is the value of the expression `5/11`. Having this table allows us to replace the inner loop, which contained an unbounded number of rolls, with a test against one call to `random.random`. The asymptotic complexity of this version of `playHand` is O(1).

The idea of replacing computation by **table lookup** has broad applicability and is frequently used when speed is an issue. Table lookup is an example of the general idea of **trading time for space**. As we saw in Chapter 13, it is is the key idea behind dynamic programming. We saw another example of this technique in our analysis of hashing: the larger the table, the fewer the collisions, and the faster the average lookup. In this case, the table is small, so the space cost is negligible.


```
def playHand(self):
    #An alternative, faster, implementation of playHand
    pointsDict = {4:1/3, 5:2/5, 6:5/11, 8:5/11, 9:2/5, 10:1/3}
    throw = rollDie() + rollDie()
    if throw == 7 or throw == 11:
        self.passWins += 1
        self.dpLosses += 1
    elif throw == 2 or throw == 3 or throw == 12:
        self.passLosses += 1
        if throw == 12:
            self.dpPushes += 1
        else:
            self.dpWins += 1
    else:
        if random.random() <= pointsDict[throw]: # point before 7
            self.passWins += 1
            self.dpLosses += 1
        else:                                    # 7 before point
            self.passLosses += 1
            self.dpWins += 1
```

Figure 16.4 Using table lookup to improve performance

16.4 Finding π

It is easy to see how Monte Carlo simulation is useful for tackling problems in which nondeterminism plays a role. Interestingly, however, Monte Carlo simulation (and randomized algorithms in general) can be used to solve problems that are not inherently stochastic, i.e., for which there is no uncertainty about outcomes.

Consider π. For thousands of years, people have known that there is a constant (called π since the 18th century) such that the circumference of a circle is equal to π*diameter and the area of the circle equal to $\pi * \text{radius}^2$. What they did not know was the value of this constant.

One of the earliest estimates, $4*(8/9)^2 = 3.16$, can found in the Egyptian *Rhind Papyrus*, circa 1650 BC. More than a thousand years later, the *Old Testament* implied a different value for π when giving the specifications of one of King Solomon's construction projects,

And he made a molten sea, ten cubits from the one brim to the other: it was round all about, and his height was five cubits: and a line of thirty cubits did compass it round about.[116]

Solving for π, $10\pi = 30$, so $\pi = 3$. Perhaps the *Bible* is simply wrong, or perhaps the molten sea wasn't perfectly circular, or perhaps the circumference was measured from the outside of the wall and the diameter from the inside, or perhaps it's just poetic license. We leave it to the reader to decide.

Archimedes of Syracuse (287-212 BCE) derived upper and lower bounds on the value of π by using a high-degree polygon to approximate a circular shape. Using a polygon with 96 sides, he concluded that $223/71 < \pi < 22/7$. Giving upper and lower bounds was a rather sophisticated approach for the time. Also, if we take his best estimate as the average of his two bounds we obtain 3.1418, an error of about 0.0002. Not bad!

Long before computers were invented, the French mathematicians Buffon (1707-1788) and Laplace (1749-1827) proposed using a stochastic simulation to estimate the value of π.[117] Think about inscribing a circle in a square with sides of length 2, so that the radius, r, of the circle is of length 1.

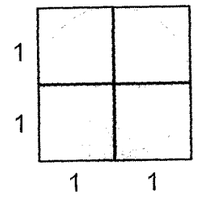

Figure 16.5 Unit circle inscribed in a square

By the definition of π, $area = \pi r^2$. Since r is 1, π = area. But what's the area of the circle? Buffon suggested that he could estimate the area of a circle by a dropping a large number of needles (which he argued would follow a random path as they fell) in the vicinity of the square. The ratio of the number of needles with tips lying within the square to the number of needles with tips lying within the circle could then be used to estimate the area of the circle.

[116] King James Bible, 1 Kings 7.23.

[117] Buffon proposed the idea first, but there was an error in his formulation that was later corrected by Laplace.

If the locations of the needles are truly random, we know that

$$\frac{\text{needles in circle}}{\text{needles in square}} = \frac{\text{area of circle}}{\text{area of square}}$$

and solving for the area of the circle,

$$\text{area of circle} = \frac{\text{area of square} * \text{needles in circle}}{\text{needles in square}}$$

Recall that the area of a 2 by 2 square is 4, so,

$$\text{area of circle} = \frac{4 * \text{needles in circle}}{\text{needles in square}}$$

In general, to estimate the area of some region R

1. Pick an enclosing region, E, such that the area of E is easy to calculate and R lies completely within E.
2. Pick a set of random points that lie within E.
3. Let F be the fraction of the points that fall within R.
4. Multiply the area of E by F.

If you try Buffon's experiment, you'll soon realize that the places where the needles land are not truly random. Moreover, even if you could drop them randomly, it would take a very large number of needles to get an approximation of π as good as even the *Bible*'s. Fortunately, computers can randomly drop simulated needles at a ferocious rate.

Figure 16.6 contains a program that estimates π using the Buffon-Laplace method. For simplicity, it considers only those needles that fall in the upper right-hand quadrant of the square.

The function `throwNeedles` simulates dropping a needle by first using `random.random` to get a pair of positive Cartesian coordinates (x and y values) representing the position of the needle with respect to the center of the square. It then uses the Pythagorean theorem to compute the hypotenuse of the right triangle with base x and height y. This is the distance of the tip of the needle from the origin (the center of the square). Since the radius of the circle is 1, we know that the needle lies within the circle if and only if the distance from the origin is no greater than 1. We use this fact to count the number of needles in the circle.