MODULE 9.5

Random Walk

Downloads

For several computational tools, the text's website has a *RandomWalk* file, which contains code for the module's algorithms, available for download. The site also has a data file *AverageDistances.dat* for Project 3.

Introduction

One technique of Monte Carlo simulations that has many applications in the sciences is the random walk. **Random walk** refers to the apparently random movement of an entity. In a time-driven simulation, we depict the entity in a **cell** on a rectangular **grid**. At any time step, the entity can move, perhaps under certain constraints, at random to a neighboring cell.

Definition **Random walk** refers to the apparently random movement of an entity.

A certain type of computer simulation involving grids is a cellular automaton. **Cellular automata** are dynamic computational models that are discrete in space, state, and time. We picture space as a one-, two-, or three-dimensional **grid**, or array, or lattice. A **site**, or **cell**, of the grid has a state, and the number of states is finite. **Rules**, or **transition rules**, specifying local relationships and indicating how cells are to change state, regulate the behavior of the system. An advantage of such grid-based models is that we can visualize the progress of events through informative animations. For example, we can view a simulation of the movement of ants toward a food source, the spread of fire, or the motion of gas molecules in a container. In this module, the next chapter, and various modules in Chapter 14, we consider many scientific applications involving cellular automata.

Definitions A **cellular automaton** (plural, **automata**) is a type of computer simulation that is a dynamic computational model and is discrete in space, state, and time. Space is a **grid**, or a one-, two-, or three-dimensional lattice, or array, of **sites**, or **cells**. A cell of the lattice has a state, and the number of states is finite. **Rules**, or **transition rules**, specifying local relationships and indicating how cells are to change state, regulate the behavior of the system.

A random walk cellular automaton can model **Brownian motion**, which is the behavior of a molecule suspended in a liquid. The phenomenon bears the name of the English botanist Robert Brown. In 1827, he observed the rapid, random motion of pollen particles in a liquid could not occur because of life within the pollen, as some conjectured. A generation later, the physicists Maxwell, Clausius, and Einstein explained the phenomenon as invisible liquid particles striking the visible particles, causing small movements. Because diffusion of many things, such as pollutants in the atmosphere and calcium in living bone tissue, exhibit Brownian motion, simulations using random walks can also model these processes (Encyclopedia Britannica 1997; Exploratorium 1995).

In genetics, random walks have been used to simulate mutation of genes. As another example, scientists use the method **polymerase chain reaction** (**PCR**) to make many copies of particular pieces of DNA. A strand of DNA contains sequences of four bases, A, T, C, and G. Using the random walk technique in simulations, computational scientists can determine good proportions of these bases in solution to speed replication of the DNA.

Algorithm for Random Walk

At each time step of a particular random walk simulation, suppose an entity moves in a random, diagonal direction—NE, NW, SE, or SW. To go in such a direction, the entity walks east or west one unit and north or south one unit, covering a diagonal distance of $\sqrt{2}$ units.

We develop a function, ***randomWalkPoints***, with parameter, n, for the number of steps. The function generates such a walk and returns a list or array of the coordinates of the steps. In the function body, variables x and y store the horizontal and vertical coordinates, respectively, of the current location, and variable *lst* holds a list of locations in the path of the entity. Because the walker starts at the origin, we initialize *lst* to be a list containing the point (0, 0). With parameter n being the number of steps to be taken, a loop to produce the path executes n times. Within the loop, we generate one random integer of 0 or 1 to determine if the entity turns to the east or west by incrementing or decrementing x by 1, respectively. Then, another such "flip of the coin" dictates north with an increment of y or south with a decrement. We then append the new point (x, y) onto the developing *lst*. After the loop at the end of the function, we return this list of points.

Following is pseudocode, or a structured English outline of the design, for the function *randomWalkPoints* with **left-facing arrows** (←) indicating assignment. The parameter, *n*, appears in parentheses after the function name and a description of the action of the function follows. **Preconditions**, or the conditions that must be true for the function to behave properly, appear after "Pre." Preconditions should include any assumptions and information, such as parameters and their descriptions, that the function needs to meet its objectives. **Postconditions**, which follow "Post," describe the state of the system when the function finishes executing, any error conditions, and the information the function returns or otherwise communicates.

***randomWalkPoints(n)*:**

Function to produce a random walk, where at each time step the entity goes diagonally in a NE, NW, SE, or SW direction, and to return a list of the points in the walk

```
Pre:    n is the number of steps in the walk.
Post:   A list of the points in the walk has been returned.
Algorithm:
    x ← 0 and y ← 0
    lst ← a list containing the origin
    do the following n times:
        rand ← a random 0 or 1
        if rand is 0
            increment x by 1
        else
            decrement x by 1
        rand ← a random 0 or 1
        if rand is 0
            increment y by 1
        else
            decrement y by 1
        append point (x, y) onto end of lst
    return lst
```

After calling *randomWalkPoints* to generate the list containing the points of a path, we can create and display a graphics representing the random walk. For example, we might show all the random walk locations as colored dots, the path as line segments, and the first and last points as black dots. One execution of this code displays a graphic similar to Figure 9.5.1. Because the walk is random, each run of the function *randomWalkPoints* will very probably produce a different walk.

Quick Review Question 1

The following questions refer to *randomWalkPoints:*

a. After execution of the loop, how many elements does *lst* have?

b. Is it possible for the points (3, 5) and (3, 6) to be adjacent to each other in *lst*?

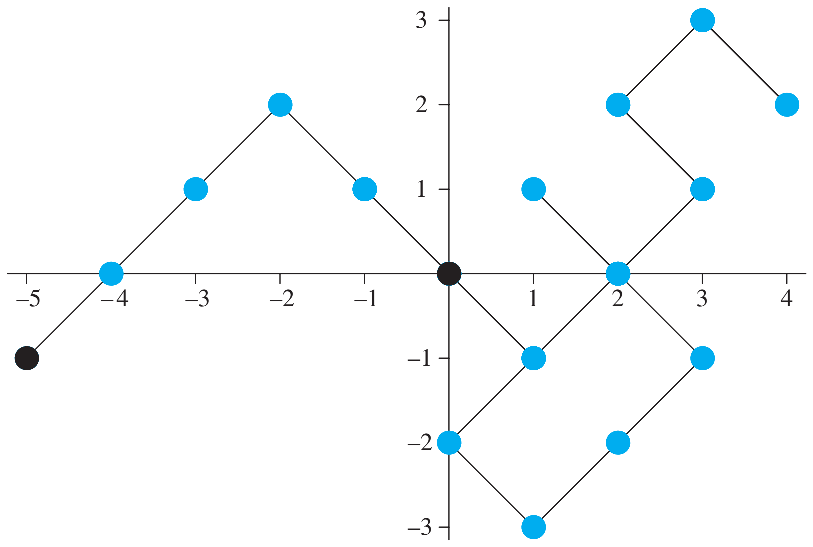

Figure 9.5.1 One possible display from execution of *randomWalkPoints*

Animate Path

Visualization of the path as it develops can aid in understanding the movement of the entity. Figure 9.5.2 presents several frames in such an animation.

To generate an animation, we develop a function, ***animateWalk***, which has as a parameter a list, *lst*, of $n + 1$ points in a random walk. For each i going from 1 through $n + 1$, we create a graphics of the first i points of the walk, which are in a sublist of the first i points of *lst*. Thus, we generate a sequence of $n + 1$ displays that we can animate with an appropriate computational tool. For the animation to be consistent, we specify that each graphics have the same axes, between the minimum and maximum of all x-coordinates on the x-axis and the minimum and maximum y-coordinates on the y-axis. The complete design of *animateWalk* follows.

***animateWalk*(*lst*)**

Function to generate an animation of a random walk

***Pre*:** *lst* is the list of the points in the walk.
***Post*:** An animation of the walk has been generated.
Algorithm:

```
xMin ← minimum of x-coordinates in lst
xMax ← maximum of x-coordinates in lst
yMin ← minimum of y-coordinates in lst
yMax ← maximum of y-coordinates in lst
for i going from 1 through n + 1 do the following:
    display a graphics of the first i points of lst with the display going from
    xMin to xMax in the x-direction and from yMin to yMax in the y-direction
```

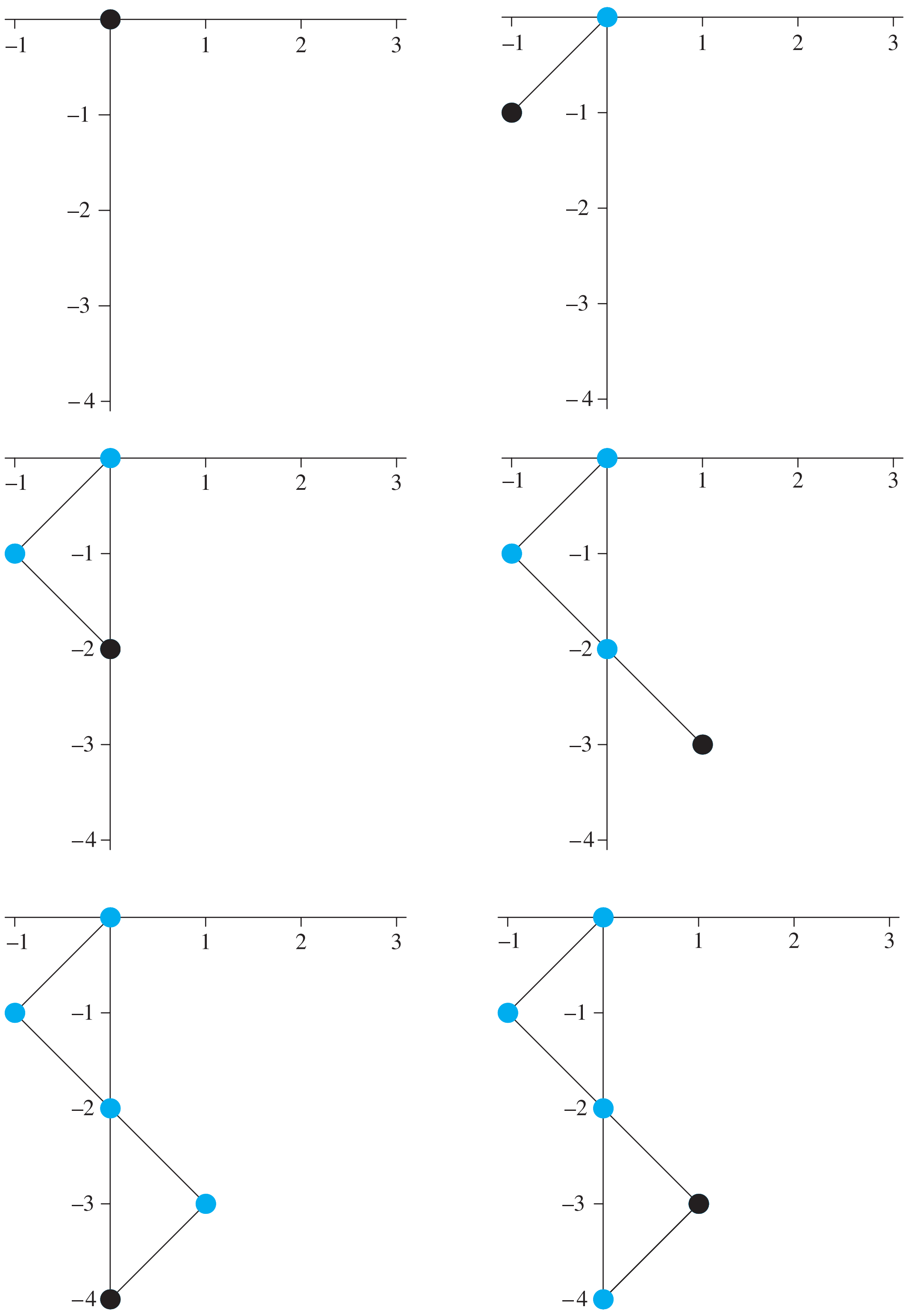

Figure 9.5.2 Several frames in an animation of the developing path from one random walk

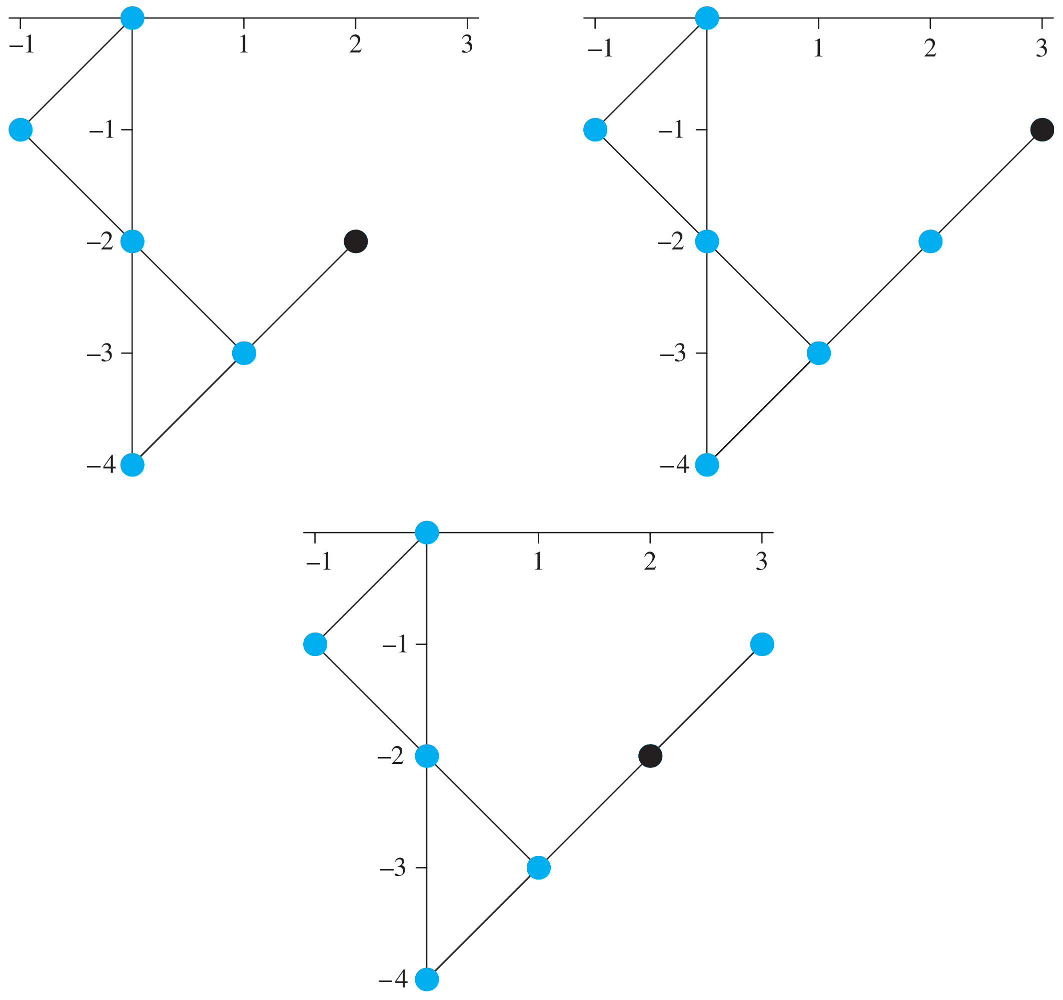

Figure 9.5.2 (*continued*)

Average Distance Covered

For the random walk in Figure 9.5.1, 5.09902 units is the distance between the final point and the initial one, which are the two black dots. However, because the walks are random, great variation can exist in both the paths and in the final distances from the starting point. Thus, to obtain an estimate of a typical distance between the starting and ending points of a random walk of n steps, we should run the simulation many times and take the average of all the distances. In such a case, we are not interested in viewing a random walk, so we first define another function, ***randomWalkDistance***, that is similar to *randomWalkPoints*, but which returns the desired distance instead of the list of points in a walk. Thus, in the loop that processes each step, we keep only the current point, (x, y), and, after the loop, we return the distance from the last point value of (x, y) and the origin, $\sqrt{x^2 + y^2}$. The next Quick Review Question designs this function.

Quick Review Question 2

Similar to *randomWalkPoints*, give a design for *randomWalkDistance*, a function with parameter, *n*, that returns the distance between the first and last point of a random walk of *n* steps.

For a function ***meanRandomWalkDistance***, which returns the average distance traveled over *numTests* number of random walks of *n* steps each, we place a call to *randomWalkPoints*(*n*) in a loop that iterates *numTests* number of times. A variable, *sumDist*, accumulates the distances covered by the random walks. Before the loop, *sumDist* is initialized to zero; after the loop, this sum is divided by *numTests* to return the average distance. One run of *meanRandomWalkDistance*(25, 100) might return an average distance of 5.75278 units for 100 simulations of random walks of 25 steps. The design of the function follows.

***meanRandomWalkDistance*(*n*, *numTests*)**

Function to run a random walk simulation *numTests* number of times and to return the average distance between the first and last points

```
Pre:  n is the number of steps in a walk.
      numTests is the number of times to run the simulation.
Post: The average distance between the first and last points has been
      returned.
Algorithm:
    let sumDist, the ongoing sum of distances, be 0
    do the following numTests times:
        add randomWalkDistance(n) to sumDist
    return sumDist / numTests as a floating point number
```

Quick Review Question 3

If we incorrectly move the initialization of *sumDist* inside the outer loop of *meanRandomWalkDistance*, select the final value of *sumDist:*

A. No change from current result.
B. *sumDist* would be 0.
C. *sumDist* would hold only the distance for the final path.
D. *sumDist* would be undefined.

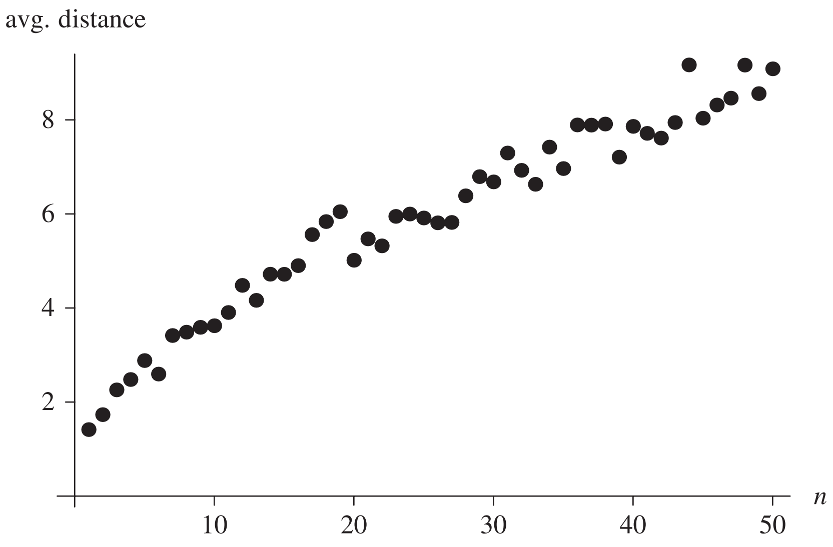

Figure 9.5.3 A plot of average distances traveled versus number of steps in a random walk

Relationship between Number of Steps and Distance Covered

To discern a relationship between the number of steps, *n*, and average distance covered in a random walk, we execute *meanRandomWalkDistance*(*n*, 100) for values of *n* from 1 to 50 and store each average distance in a list or array, *listDist*. Then, we employ the techniques of Module 8.3, "Empirical Models," to determine the relationship. Figure 9.5.3 shows a plot of the average distances traveled versus the number of steps. Projects 3 and 4 determine a formula for this relationship.

Exercises

On the text's website, RandomWalk files for several computational tools contain the code for the functions of this module. Complete the following exercises using your computational tool.

1. If possible in your computational tool, revise the code of *randomWalkPoints* to replace the loop with a call to a function to formulate *lst*.

2. Revise the code of *randomWalkPoints* or Exercise 1 to have the entity go with equal probability in a N, S, E, or W direction. *Hint:* Choose the direction based the value of a random integer, 0, 1, 2, or 3.

3. a. Revise the code of *randomWalkPoints* to have the entity go in an easterly direction (incrementing *x*) with probability of 30% and in a westerly direction (decrementing *x*) with probability of 70%.

   b. Revise the code of Part a, to have the entity go in a northerly direction (incrementing *y*) with probability of 45% and in a southerly direction (decrementing *y*) with probability of 55%.
   c. Give the probability for the entity going in each direction, NE, NW, SE, and SW.
4. Revise the code of *randomWalkPoints* or Exercise 1 to have the entity go in a N, S, E, or W direction with probabilities of 20%, 30%, 45%, or 5%, respectively.

Projects

On the text's website, RandomWalk files for several computational tools contain the code for the module's algorithms. Complete the following projects using your software system.

For additional projects, see Module 14.1, "Polymers—Strings of Pearls," and Module 14.2, "Solidification—Let's Make It Crystal Clear!"

1. Exercise 1
2. Exercise 2
3. Download the data file *AverageDistances.dat* of average distances covered for step sizes from 1 to 50 from the text's website. Using the techniques of Module 8.3, "Empirical Models," determine a relationship between the number of steps, *n*, and average distance covered in a random walk.
4. Develop code as discussed in the section on "Relationship between Number of Steps and Distance Covered" to obtain a list of average distances covered for random walks of step sizes from 1 to 50. Then, using the data the program generates, do the analysis of Project 3.
5. Develop code as discussed in the section on "Relationship between Number of Steps and Distance Covered" to obtain a list of average distances covered for random walks of step sizes from 1 to 50, where the entity travels E, W, N, or S with each step. Then, using the data the program generates, do the analysis of Project 3.
6. Develop code for Exercise 3 and run the simulation for 50 time steps. Include this code in a loop that runs the simulation 1000 or more times. Have the segment return the portion of time the entity ends on the 50th step in each of the four quadrants, NE, NW, SE, and SW. Do the figures seem to agree with your answer to Exercise 3c?
7. Develop code for Exercise 4 and run the simulation for 50 time steps. Include this code in a loop that runs the simulation 1000 or more times. Have the segment return the portion of time the entity ends on the 50th step in the N, S, E, or W direction from the starting location, the origin. On a particular run of the simulation, the 50th step could fall into one category, such as due north of the origin, or in two categories, such as N and E of the origin. Discuss the results in relationship to the probabilities of Exercise 4.
8. A hiker without a compass trying to find the way in the dark can step in any of eight directions (N, NE, E, SE, S, SW, W, NW) with each step. Studies

show that people tend to veer to the right under such circumstances. Initially, the hiker is facing north. Suppose at each step probabilities of going in the indicated directions are as follows: N, 19%; NE, 24%; E, 17%; SE, 10%; S, 2%; SW, 3%; W, 10%; NW, 15%. Develop a simulation to trace a path of a hiker, and run the simulation a number of times. Describe the results. (Note that other than at the initial step, this simulation simplifies the problem by ignoring the direction in which the hiker faces.)

9. Perform a simulation of Brownian motion of a pollen grain suspended in a liquid by generating a 3D random walk. Using documentation for your computational tool, investigate how to plot 3D graphics points and lines and create a 3D graphic of the walk.

Answers to Quick Review Question

1. a. $n + 1$ elements, (0, 0) and the n appended points
b. No, both coordinates are changed in the body of the loop.

2.

***randomWalkDistance(n)*:**

Function to produce a random walk, where at each time step the entity goes diagonally, and to return the distance between the first and last points

```
Pre:   n the number of steps in the walk.
Post:  The distance between the first and last points of a random walk of n steps was returned.
Algorithm:
    $x \leftarrow 0$ and $y \leftarrow 0$
    do the following $n$ times:
        $rand \leftarrow$ a random 0 or 1
        if $rand$ is 0, increment $x$ by 1; else decrement $x$ by 1
        $rand \leftarrow$ a random 0 or 1
        if $rand$ is 0, increment $y$ by 1; else decrement $y$ by 1
    return $\sqrt{x^2 + y^2}$
```

3. C. *sumDist* would hold only the distance for the final path.

References

Encyclopedia Britannica. 1997. "Brownian Motion." *Britannica Guide to the Nobel Prizes. Britannica Online*. http://www.britannica.com/nobel/micro/88_96.html

Exploratorium. 1995. "Brownian Motion." *Exploratorium Exhibit and Phenomena Cross-Reference*. http://www.exploratorium.edu/xref/phenomena/brownian_motion.html