Observations Our first challenge is to determine how to make quantitative measurements of the running time of our programs. This task is far easier than in the natural sciences. We do not have to send a rocket to Mars or kill laboratory animals or split an atom—we can simply run the program. Indeed, *every* time you run a program, you are performing a scientific experiment that relates the program to the natural world and answers one of our core questions: *How long will my program take?*

Our first qualitative observation about most programs is that there is a *problem size* that characterizes the difficulty of the computational task. Normally, the problem size is either the size of the input or the value of a command-line argument. Intuitively, the running time should increase with problem size, but the question of *by how much* it increases naturally comes up every time we develop and run a program.

Another qualitative observation for many programs is that the running time is relatively insensitive to the input itself; it depends primarily on the problem size. If this relationship does not hold, we need to take steps to better understand and perhaps better control the running time's sensitivity to the input. But it does often hold, so we now focus on the goal of better quantifying the relationship between problem size and running time.

Example. As a running example, we will work with the program `ThreeSum` shown here, which counts the number of (unordered) *triples* in a file of `n` integers that sum to 0 (assuming that overflow plays no role). For simplicity, we assume that the `n` integers are distinct, so that each triple refers to three distinct integers. This computation may seem contrived to you, but it is deeply related to numerous fundamental computational tasks (for example, see EXERCISE 1.4.26). As a test input, consider the file `1Mints.txt` from the booksite, which contains 1 million randomly generated `int` values. The second, eighth, and tenth entries in `1Mints.txt` sum to 0. How many more such triples are there in the file? `ThreeSum` can tell us, but can it do so in a reasonable amount of time? What is the relationship between the problem size `n` and running time for `ThreeSum`? As a first experiment, try running `ThreeSum`

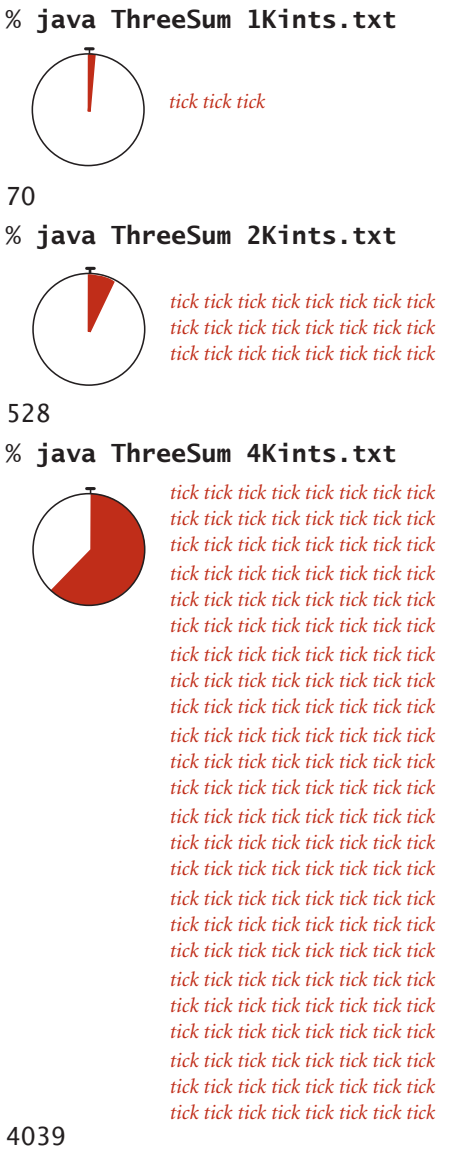
```
public class ThreeSum
{
   public static int count(int[] a)
   {  // Count triples that sum to 0.
      int n = a.length;
      int count = 0;
      for (int i = 0; i < n; i++)
         for (int j = i+1; j < n; j++)
            for (int k = j+1; k < n; k++)
               if (a[i] + a[j] + a[k] == 0)
                  count++;
      return count;
   }

   public static void main(String[] args)
   {
      In in = new In(args[0]);
      int[] a = in.readAllInts();
      StdOut.println(count(a));
   }
}
```

Given N, how long will this program take?

```
% more 1Mints.txt
 324110
-442472
 626686
-157678
 508681
 123414
 -77867
 155091
 129801
 287381
 604242
 686904
-247109
  77867
 982455
-210707
-922943
-738817
  85168
 855430
 ...
```

on your computer for the files `1Kints.txt`, `2Kints.txt`, `4Kints.txt`, and `8Kints.txt` on the booksite that contain the first 1,000, 2,000, 4,000, and 8,000 integers from `1Mints.txt`, respectively. You can quickly determine that there are 70 triples that sum to 0 in `1Kints.txt` and that there are 528 triples that sum to 0 in `2Kints.txt`. The program takes substantially more time to determine that there are 4,039 triples that sum to 0 in `4Kints.txt`, and as you wait for the program to finish for `8Kints.txt`, you will find yourself asking the question *How long will my program take*? As you will see, answering this question for this program turns out to be easy. Indeed, you can often come up with a fairly accurate prediction while the program is running.



Observing the running time of a program

Stopwatch. Reliably measuring the exact running time of a given program can be difficult. Fortunately, we are usually happy with estimates. We want to be able to distinguish programs that will finish in a few seconds or a few minutes from those that might require a few days or a few months or more, and we want to know when one program is twice as fast as another for the same task. Still, we need accurate measurements to generate experimental data that we can use to formulate and to check the validity of hypotheses about the relationship between running time and problem size. For this purpose, we use the `Stopwatch` data type shown on the facing page. Its `elapsedTime()` method returns the elapsed time since it was created, in seconds. The implementation is based on using the Java system's `currentTimeMillis()` method, which gives the current time in milliseconds, to save the time when the constructor is invoked, then uses it again to compute the elapsed time when `elapsedTime()` is invoked.

API

public class Stopwatch	
Stopwatch()	*create a stopwatch*
double elapsedTime()	*return elapsed time since creation*

typical client

```
public class StopwatchClient
{
   public static void main(String[] args)
   {
      int n = Integer.parseInt(args[0]);
      int[] a = new int[n];
      for (int i = 0; i < n; i++)
         a[i] = StdRandom.uniform(-1000000, 1000000);
      Stopwatch timer = new Stopwatch();
      int count = ThreeSum.count(a);
      double time = timer.elapsedTime();
      StdOut.println(count + " triples " + time + " seconds");
   }
}
```

application

```
% java StopwatchClient 1000
51 triples 0.488 seconds

% java StopwatchClient 2000
516 triples 3.855 seconds
```

implementation

```
public class Stopwatch
{
   private final long start;

   public Stopwatch()
   {  start = System.currentTimeMillis();  }

   public double elapsedTime()
   {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }

}
```

An abstract data type for a stopwatch

Analysis of experimental data. The program `DoublingTest` on the facing page is a more sophisticated `Stopwatch` client that produces experimental data for `ThreeSum`. It generates a sequence of random input arrays, doubling the array size at each step, and prints the running times of `ThreeSum.count()` for each input size. These experiments are certainly reproducible—you can also run them on your own computer, as many times as you like. When you run `DoublingTest`, you will find yourself in a prediction-verification cycle: it prints several lines very quickly, but then slows down considerably. Each time it prints a line, you find yourself wondering how long it will be until it prints the next line. Of course, since you have a different computer from ours, the actual running times that you get are likely to be different from those shown for our computer. Indeed, if your computer is twice as fast as ours, your running times will be about half ours, which leads immediately to the well-founded hypothesis that running times on different computers are likely to differ by a constant factor. Still, you will find yourself asking the more detailed question *How long will my program take, as a function of the input size?* To help answer this question, we plot the data. The diagrams at the bottom of the facing page show the result of plotting the data, both on a normal and on a log-log scale, with the problem size n on the x-axis and the running time $T(n)$ on the y-axis. The log-log plot immediately leads to a hypothesis about the running time—the data fits a straight line of slope 3 on the log-log plot. The equation of such a line is

$$\lg(T(n)) = 3 \lg n + \lg a$$

(where a is a constant) which is equivalent to

$$T(n) = a n^3$$

the running time, as a function of the input size, as desired. We can use one of our data points to solve for a—for example, $T(8{,}000) = 51.1 = a\,8000^3$, so $a = 9.98 \times 10^{-11}$—and then use the equation

$$T(n) = 9.98 \times 10^{-11} n^3$$

to predict running times for large n. Informally, we are checking the hypothesis that the data points on the log-log plot fall close to this line. Statistical methods are available for doing a more careful analysis to find estimates of a and the exponent b of n, but our quick calculations suffice to estimate running time for most purposes. For example, we can estimate the running time on our computer for $n = 16{,}000$ to be about $9.98 \times 10^{-11} \times 16{,}000^3 = 408.8$ seconds, or about 6.8 minutes (the actual time was 409.3 seconds). While waiting for your computer to print the line for $n = 16{,}000$ in `DoublingTest`, you might use this method to predict when it will finish, then check the result by waiting to see if your prediction is true.

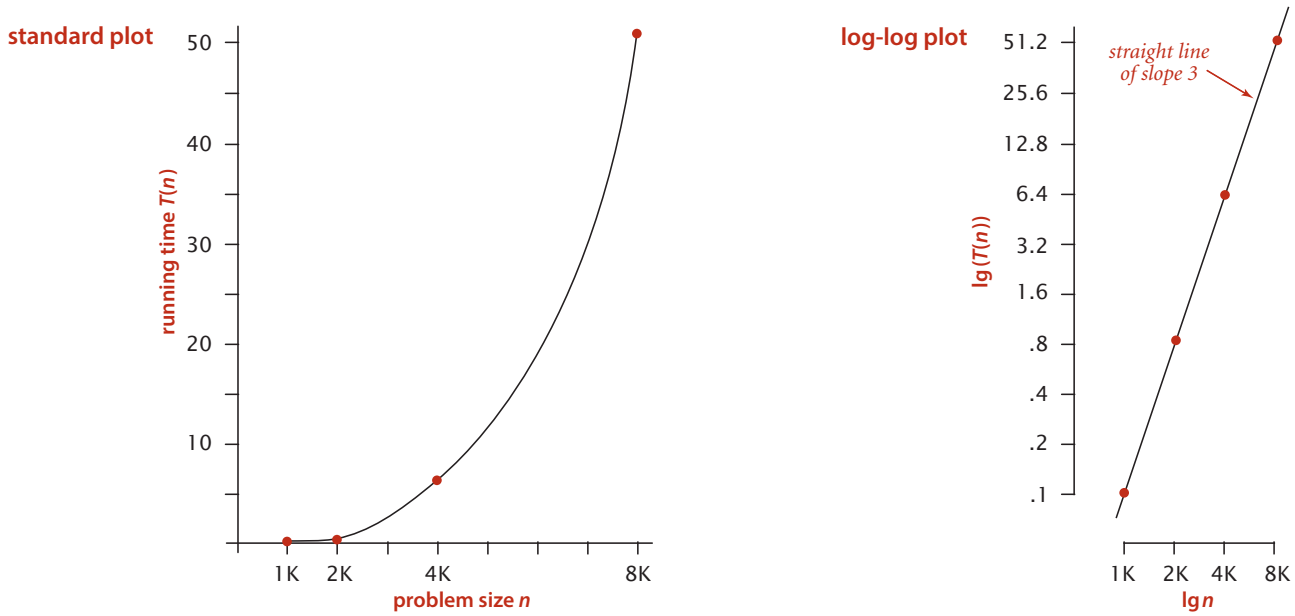
program to perform experiments

```
public class DoublingTest
{
   public static double timeTrial(int n)
   {  // Time ThreeSum.count() for n random 6-digit ints.
      int MAX = 1000000;
      int[] a = new int[n];
      for (int i = 0; i < n; i++)
         a[i] = StdRandom.uniform(-MAX, MAX);
      Stopwatch timer = new Stopwatch();
      int count = ThreeSum.count(a);
      return timer.elapsedTime();
   }

   public static void main(String[] args)
   {  // Print table of running times.
      for (int n = 250; true; n *= 2)
      {  // Print time for problem size n.
         double time = timeTrial(n);
         StdOut.printf("%7d %7.1f\n", n, time);
      }
   }
}
```

results of experiments

```
% java DoublingTest
    250     0.0
    500     0.0
   1000     0.1
   2000     0.8
   4000     6.4
   8000    51.1
...
```



Analysis of experimental data (the running time of `ThreeSum.count()`)

So far, this process mirrors the process scientists use when trying to understand properties of the real world. A straight line in a log-log plot is equivalent to the hypothesis that the data fits the equation $T(n) = a\,n^b$. Such a fit is known as a *power law*. A great many natural and synthetic phenomena are described by power laws, and it is reasonable to hypothesize that the running time of a program does, as well. Indeed, for the analysis of algorithms, we have mathematical models that strongly support this and similar hypotheses, to which we now turn.

Mathematical models In the early days of computer science, D. E. Knuth postulated that, despite all of the complicating factors in understanding the running times of our programs, it is possible, in principle, to build a mathematical model to describe the running time of any program. Knuth's basic insight is simple: the total running time of a program is determined by two primary factors:

- The cost of executing each statement
- The frequency of execution of each statement

The former is a property of the computer, the Java compiler and the operating system; the latter is a property of the program and the input. If we know both for all instructions in the program, we can multiply them together and sum for all instructions in the program to get the running time.

The primary challenge is to determine the frequency of execution of the statements. Some statements are easy to analyze: for example, the statement that sets `count` to 0 in `ThreeSum.count()` is executed exactly once. Others require higher-level reasoning: for example, the `if` statement in `ThreeSum.count()` is executed precisely

$$n\,(n-1)\,(n-2)\,/\,6$$

times (the number of ways to pick three different numbers from the input array—see EXERCISE 1.4.1). Others depend on the input data: for example the number of times the instruction `count++` in `ThreeSum.count()` is executed is precisely the number of triples that sum to 0 in the input, which could range from 0 of them to all of them. In the case of `DoublingTest`, where we generate the numbers randomly, it is possible to do a probabilistic analysis to determine the expected value of this quantity (see EXERCISE 1.4.40).

Tilde approximations. Frequency analyses of this sort can lead to complicated and lengthy mathematical expressions. For example, consider the count just considered of the number of times the `if` statement in `ThreeSum` is executed:

$$n\,(n-1)\,(n-2)\,/\,6 = n^3/\,6 - n^2/\,2 + n\,/\,3$$