1

SPECIAL NUMBERS: TRIANGULAR, OBLONG, PERFECT, DEFICIENT, AND ABUNDANT

We start our introduction to number theory with definitions, properties, and relationships of several categories of numbers.

TRIANGULAR NUMBERS

Triangular numbers are those that can be written as the sum of a consecutive series of (whole) numbers beginning with 1. Thus 6 is triangular because it is the sum of the first three numbers: $6 = 1 + 2 + 3$. The first few triangular numbers are 1, 3, 6, 10, 15, 21, 28, 36, 45, and 55. We denote the nth triangular number by t_n. Thus $t_5 = 1 + 2 + 3 + 4 + 5 = 15$. More generally,

$$t_n = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \tag{1.1}$$

Our first program, calculating a specific triangular number, shows the format of an HTML document. The first line specifies the **doctype**. The rest is an *html* element, starting with **<html>** and ending with **</html>**. Within the html element is a *head* element and a *body*

Elementary Number Theory with Programming, First Edition. Marty Lewinter and Jeanine Meyer.

element. In this case, the body element is empty. The head element contains a **meta** tag specifying the character type (it can be omitted), a title, and a *script* element. All the action is in the script element.

The code makes use of standard programming constructs such as variables and functions and for-loops (if you don't understand what these terms are, please consult any beginner book on programming. Shameless plug: go to *The Essential Guide to HTML5: Using Games to Learn HTML5 and JavaScript*, http://www.apress.com/9781430233831).

The specific triangular number we want is specified in the coding by setting the variable **n**. This is termed *hard-coding*. The computation is done using a for-loop. The for-loop adds up the values from `1 to n`, exactly following Equation 1.1. The built-in method `document.write` writes out the result.

The challenge in Exercise 1 is to compare coding using Equation 1.1 versus Equation 1.2. The challenge is that computers are very fast. I use the built-in `Date` function with the method `getTime` to get the number of milliseconds from a base date at the start and after the computation. It turns out that computing the millionth triangular number takes 3 ms! You can experiment with different values. Using the formula given in Equation 1.2 would be much, much faster. Give it a try.

The nth triangular number is given by the formula:

$$t_n = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \tag{1.2}$$

Example: $t_{100} = \dfrac{100 \times 101}{2} = 5{,}050$

Example: Write $6 + 7 + 8 + 9 + 10 + 11$ as the difference of two triangular numbers. We observe that $6 + 7 + 8 + 9 + 10 + 11 = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11) - (1 + 2 + 3 + 4 + 5)$, which is $t_{11} - t_5$.

Example: Generalize the previous example to any consecutive sum such as $45 + 46 + \cdots + 987$. Note that $a + (a+1) + (a+2) + \cdots + b = (1 + 2 + 3 + \cdots + b) - (1 + 2 + 3 + \cdots + (a-1)) = t_b - t_{a-1}$. By letting $a = 6$ and $b = 11$, we get the result of the previous example.

It should be noted that

$$t_n - t_{n-1} = n \tag{1.3}$$

The sum of any two consecutive triangular numbers is a square. For example, $t_4 + t_3 = 10 + 6 = 16 = 4^2$ and $t_5 + t_4 = 15 + 10 = 25 = 5^2$. This is expressed by the formula

$$t_n + t_{n-1} = n^2 \tag{1.4}$$

Example: Verify (1.4) for $n = 10$. We have $t_{10} + t_9 = 55 + 45 = 10^2$.

Example: Find two triangular numbers whose sum is 900. Since $900 = 30^2$, we have $n = 30$. Then using (1.4), $900 = t_{30} + t_{29} = \frac{30 \times 31}{2} + \frac{29 \times 30}{2} = 465 + 435$.

The sum of the reciprocals of all the triangular numbers is 2. Formally,

$$\frac{1}{1} + \frac{1}{3} + \frac{1}{6} + \frac{1}{10} + \cdots + \frac{1}{t_n} + \cdots = 2 \tag{1.5}$$

OBLONG NUMBERS AND SQUARES

A positive integer of the form $n(n + 1)$ is called *oblong*. The nth oblong number is the sum of the first n even numbers. To see this, observe that the nth even number is $2n$. Then we have $2 + 4 + 6 + \cdots + 2n = 2(1 + 2 + \cdots + n) = 2\left(\frac{n(n+1)}{2}\right) = n(n + 1)$, the nth oblong number. What about the sum of the first n odd numbers? The nth odd number is $2n - 1$. So $1 + 3 + 5 + \cdots + (2n-1) = (2 \times 1 - 1) + (2 \times 2 - 1) + (2 \times 3 - 1) + \cdots + (2n - 1)$, in which -1 appears n times. We then get $2(1 + 2 + 3 + \cdots + n) - n = 2\left(\frac{n(n+1)}{2}\right) - n = n(n + 1) - n = n^2 + n - n = n^2$. So the sum of the first n odd numbers is n^2.

Example: The sum of the first 5 odd numbers is 25. (Check this: $1 + 3 + 5 + 7 + 9 = 25$.) More impressively, the sum of the first 100 odd numbers is $100^2 = 10{,}000$.

The great French mathematician LaGrange (1736–1813) showed in the late eighteenth century that every positive number can be written as a sum of four or fewer squares. Thus, for example, $30 = 25 + 4 + 1$.

Number theorists are fond of numbers, such as 40, which are the sum of only two squares (e.g., $40 = 36 + 4$).

The Pythagoreans computed the sum of the first n powers of 2. Let

a. $S = 1 + 2 + 4 + \cdots + 2^{n-1}$. Then

b. $2S = 2 + 4 + \cdots + 2^{n-1} + 2^n$.

Now subtract Equation (a) from Equation (b), and we get $S = 2^n - 1$. We have, then, the following formula:

$$1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1} = 2^n - 1 \tag{1.6}$$

With a minor change in the proof of (1.6), we obtain an analogous formula for the sum of the first n powers of any base. Let $S = 1 + a + a^2 + a^3 + \cdots + a^{n-1}$. Then $aS = a + a^2 + a^3 + \cdots + a^{n-1} + a^n$. Subtract the first equation from the second, and we get $(a - 1)S = a^n - 1$. Upon division by $a - 1$, we obtain the following formula:

$$1 + a + a^2 + a^3 + \cdots + a^{n-1} = \frac{a^n - 1}{a - 1} \tag{1.7}$$

DEFICIENT, ABUNDANT, AND PERFECT NUMBERS

The Pythagoreans classified all numbers as *deficient*, *abundant,* or *perfect*. Given a number, find all of its *proper* factors, that is, all numbers that go into it (with the exclusion of the given number). The proper factors of 30, for example, are 1, 2, 3, 5, 6, 10, and 15.

Generating a list of the factors of a number is easy in JavaScript (and other programming languages), though it appears tedious to us. The modulo operation, %, determines the remainder. So if `n` is the number and `f` is a candidate factor, then

```
n % f
```

will produce the remainder of `n` divided by `f`. If this is 0, then `f` is a factor. If `f < n`, then `f` is a proper factor.

The program uses a for-loop going from `1` up to but not including `n`. If it is a factor, the number is written out in the html document using `document.write` and a variable `count` is incremented.

If the sum of the proper factors of n is less than n, we call n *deficient*. If the sum exceeds n, it is called *abundant*. If the sum equals n, we call it *perfect*. For example, 8 is deficient since $1+2+4<8$, 18 is abundant since $1+2+3+6+9>18$, and 28 is perfect since $1+2+4+7+14=28$. The smallest perfect number is 6. The first few perfect numbers are 6, 28, 496, and 8128. It is not known today whether there are infinitely many perfect numbers. Moreover, all known perfect numbers are even. No one knows if there are any odd perfect numbers! Incidentally, the smallest abundant odd number is 945, while the smallest abundant even number is 12.

The program to characterize a number as deficient, perfect, or abundant was made by modifying the previous one that listed and counted the number of proper factors. To make the determination of whether a number n is deficient, perfect, or abundant, the program has to add up the proper factors. So the statement count++ is removed and the statement

```
sum += i;
```

is inserted. By the way, this is shorthand for taking the original value of the variable `sum`, adding one to it and then assigning that back to the variable `sum`.

```
sum = sum + i;
```

I also changed the name of the function to `addUpFactors`. I tested the program using the specific numbers given in the text.

The Pythagoreans found an amazing method for finding perfect numbers. They observed, using (1.6), that sums of the form $1+2+2^2+2^3+\cdots+2^{n-1}$ are *prime* for certain values of n and are *composite* for others. (A number is *prime* if its only factors are 1 and itself. 7, 19, and 31 are examples of primes. A *composite* number has proper factors other than 1. Thus 20 is composite.) The following sums, for example, are prime:

$$1+2=3$$

$$1+2+4=7$$

$$1+2+4+8+16=31$$

$$1+2+4+8+16+32+64=127$$

In each of these equations, multiply the greatest number on the left by the number on the right, yielding $2\times 3=6$, $4\times 7=28$, $16\times 31=496$, and $64\times 127=8128$. These products, 6, 28, 496, and 8128 are perfect. Whenever the sum of the first n powers of 2 is prime, this procedure yields a perfect number! Using (1.6), the sum of the first n powers of 2 is 2^n-1, so the perfect number is of the form $2^{n-1}(2^n-1)$. The prime sum, 2^n-1, is then called a *Mersenne prime*, in honor of the eighteenth century French mathematician. It was shown in the eighteenth century by the great Swiss mathematician Leonhard Euler (1707–1783) that all even perfect numbers are of the form $2^{n-1}(2^n-1)$.

A conjecture is that no odd number (odd number >1) is perfect. One of the exercises and one of our programs tests this conjecture on the first 1000 odd numbers.

The program to test the conjecture concerning odd numbers not being perfect numbers is built on the previous example. Instead of writing out the result, a function with a parameter is made to return –1, 0, or 1 if the number is deficient, perfect, or abundant. Any three distinct values could be used.

The inner function (I named it `classify`) is called for all odd numbers up to a limit. The task is to determine how to generate the set of numbers. A solution is to use a for-loop going from `j=1 to 1000` and, within the loop, setting a variable `n` to `2*j+1`.

It is very important to keep in mind that this is not a proof of the conjecture. Something could be happening at higher numbers.

The Pythagoreans believed that if two friends wore amulets, one with 220 and the other with 284, they would fortify their friendship. This is because the sum of the proper factors of either one of these numbers equals the other number, that is,

$$220=1+2+4+71+142$$

$$284=1+2+4+5+10+11+20+22+44+55+110$$

We call a pair of numbers with this property, *amicable numbers*. 1184 and 1210 comprise the next pair of amicable numbers, since

$$1210 = 1 + 2 + 4 + 8 + 16 + 32 + 37 + 74 + 148 + 296 + 592$$
$$1184 = 1 + 2 + 5 + 10 + 11 + 22 + 55 + 110 + 121 + 242 + 605$$

EXERCISES

An asterisk (∗) indicates that the exercise can be developed into a research project.

1 Write a program to find the nth triangular number, t_n, using formula (1.1). Then write a program using (1.2). Compare the two procedures for very large values of n. *A program for the first part of this is included at the end of the chapter.*

2 Write a program to find out whether a given number is a square.

3 Find, using a partial fraction decomposition, the sum of the reciprocals of the first n triangular numbers, that is, find $\frac{1}{1} + \frac{1}{3} + \frac{1}{6} + \frac{1}{10} + \cdots + \frac{1}{t_n}$. Then write a program to do this.

4 *The ancient Egyptians expressed every proper fraction except as the sum of fractions with 1s in the numerator. Thus, $\frac{2}{3}$ is equivalent to $\frac{1}{3} + \frac{1}{4} + \frac{1}{12}$ and $\frac{7}{8}$ is equivalent to as $\frac{1}{2} + \frac{1}{4} + \frac{1}{8}$.

a. Verify the identity $\frac{1}{n} = \frac{1}{(n+1)} + \frac{1}{(n(n+1))}$.

b. Using the strategy of starting off by writing a fraction a/b as the sum of 1/b + 1/b + · · · (a times) and using the identity verified in part a repeatedly until all fractions are distinct, write a program to express every fraction as the sum of fractions with distinct denominators and numerators equal to 1.

5 *Note that the sum and difference of the triangular numbers 15 and 21 are triangular. Verify that this is also the case for the triangular numbers 780 and 990. Find 100 more cases.

6 Find 100 triangular numbers that are squares.

7 From the first 1000 triangular number, find ones that are the sum of two other triangular numbers.

8 *Find 100 oblong numbers that are products of an oblong number and a square.

9 Show that after 3, the next 100 triangular numbers are composite (not prime). Then prove this for all triangular numbers after 3.

10 Write each of the numbers from 1 to 1000 as the sum of three or fewer triangular numbers.

11 *Write each of the numbers from 1 to 1000 as the sum of four or fewer squares. For which of these numbers can this be done in more than one way? For example, $50 = 49 + 1 = 25 + 25 = 36 + 9 + 4 + 1 = 16 + 16 + 9 + 9$.

12 *What proportion of the first 1000 numbers can be written using two or fewer squares?

13 Write a program that lists the proper divisors of a given number. *A program for this is given at the end of the chapter.*

14 Write a program to find the sum of the proper divisors of a given number.

15 Modify the program of the previous exercise to decide whether a given number is deficient, perfect, or abundant. *A program for this is included at the end of the chapter. You can improve this in various ways, including having the user enter the number. Look ahead to an example in Chapter 2 that shows how to get user input.*

16 *Write a program to check for perfect numbers within a range. You can set the endpoints of the range within the program. You can research to find a list of the known perfect numbers AND to determine the biggest integer value that can be represented in regular JavaScript.

17 *A number is called *semi-perfect* if it is the sum of *some* (but not all) of its proper divisors. 12 is the smallest semi-perfect number since $12 = 6 + 4 + 2$. Find the next 50 semi-perfect numbers.

18 *If a given number is abundant, determine if it is semi-perfect.

19 Show that 2^n is deficient for all $n \leq 25$. Then show it is deficient for all n.

20 *Verify that 945 is the smallest odd abundant number. Find the next 10 odd abundant numbers. Do they seem to be getting further apart?

21 Observe that the square of the triangular number 6 is also triangular. Verify that this does not occur for any other triangular number (except 1) up to t_{1000}.

22 It has been conjectured that no odd number is perfect. Verify this for the first 1000 odd numbers. *See last example. You can decide on other ways to present the findings and also change the limit.*

Triangular Numbers

```
<html>
<head>
<title>Triangular Numbers</title>
<script>
var n = 1000000;
var start = new Date();
start = start.getTime();

function init(){
    var sum = 0;
 for(i=1;i<=n;i++){
    sum+=i;
 }
 now = new Date();
 now = now.getTime();
 elapsed = (now - start);
 document.write("The "+n+
      "th triangular number is "+sum+".<br/>");
 document.write("Elapsed time was "+elapsed+"
 milliseconds.");
}
init();
</script>
</head>
<body>
</body>
</html>
```

Proper Factors

```
<!DOCTYPE HTML>
<html>
<head>
<title>Proper factors</title>
<script>
var n = 30;
```

```
var count = 0;
function countUpFactors(n){
  document.write("Proper factors of "+n+" are:
  <br/>");
  for (var i=1;i<n;i++){
      if ((n%i)==0){
          count++;
          document.write(i+"<br/>");
      }
  }
  document.write("The number of proper factors
  of "+n+" is "+count+".");
}
countUpFactors(n);
</script>
</head>
<body>
</body>
</html>
```

Classifying Number as Deficient, Perfect, or Abundant

```
<!DOCTYPE HTML>
<html>
<head>
<title>Perfect or </title>
<script>
function addUpFactors(n){
  document.write("Proper factors of "+n+" are:
  <br/>");
  var sum = 0;
  for (var i=1;i<n;i++){
    if ((n%i)==0){
        document.write(i+"<br/>");
        sum += i;
    }
 }

 document.write("The sum of the proper factors
 of "+n+" is "+sum+", so "+n+" is ");
 if (sum<n) {
```

```
    document.write("deficient.");
 }
 else if (sum==n) {
    document.write("perfect.");
 }
 else {
    document.write("abundant.");
 }
 document.write("<br/>");
}
addUpFactors(6);
addUpFactors(8);
addUpFactors(18);
addUpFactors(28);
addUpFactors(945);
addUpFactors(8128);
</script>
</head>
<body>
</body>
</html>
```

Checking the Conjecture on No Perfect Odd Number (up to 1000)

```
<!DOCTYPE HTML>
<html>
<head>
<title>Sort Odd numbers </title>
<script>
function classify(n){

  var sum = 0;
  for (var i=1;i<n;i++){
     if ((n%i)==0){
        //document.write(i+"<br/>");
        sum += i;
     }
  }

  if (sum<n) {
     return -1;
```

```
   }
   else if (sum==n) {
      return 0;
   }
   else {
      return 1;
   }
   document.write("<br/>");
}
 function sortOdds(limit) {
 var perfect = 0;
 var deficient = 0;
 var abundant = 0;
 for (var j=1;j<=limit;j++) {
    var n=2*j+1;
    c = classify(n);
    if (c==-1){
     deficient++;
    }
    else if (c==1) {
     abundant++;
    }
    else {
     perfect++;
    }
 }
  document.write("First "+limit+" odd numbers:
  <br/>");
  document.write ("deficient: "+deficient+
  "<br/>");
  document.write("abundant: "+abundant+
  "<br/>");
  document.write("perfect: "+perfect+
  "<br/>");
 }
 sortOdds(1000);
</script>
</head>
<body>
</body>
</html>
```