Rekursif

Rekursi adalah teknik pemrograman di mana a method (function) memanggil dirinya sendiri. Ini mungkin terdengar seperti hal aneh untuk dilakukan, atau bahkan kesalahan besar. Namun, rekursi adalah salah satu yang paling menarik, dan salah satu teknik yang paling efektif dan mengejutkan pemrograman. Seperti menarik diri Anda sendiri dengan tali sepatu Anda (Anda memang memiliki tali sepatu, bukan?), tampaknya rekursi luar biasa saat pertama kali bertemu dengannya. Namun, ternyata tidak hanya berfungsi, ia juga memberikan kerangka kerja konseptual yang unik untuk memecahkan banyak masalah.

Dalam bab ini, kami akan memeriksa banyak contoh untuk ditunjukkan berbagai macam situasi yang dapat terjadi rekursi terapan. Kami akan menghitung bilangan segitiga dan faktor, menghasilkan anagram, melakukan pencarian biner rekursif, memecahkan teka-teki Menara Hanoi, dan menyelidiki penyortiran teknik yang disebut mergesort. Applet bengkel disediakan untuk mendemonstrasikan Menara Hanoi dan mergesort.

Kami juga akan membahas kekuatan dan kelemahan rekursif, dan menunjukkan bagaimana pendekatan rekursif dapat diubah menjadi pendekatan berbasis tumpukan.

Angka Segitiga

Dikatakan bahwa Pythagorians, sekelompok ahli matematika di Yunani kuno yang bekerja di bawah Pythagoras (dari Ketenaran teorema Pythagorian), merasakan hubungan mistik dengan deretan angka 1, 3, 6, 10, 15, 21,… (dimana … Berarti seri berlanjut tanpa batas waktu). Bisakah kamu temukan
anggota selanjutnya dari seri ini? Suku ke-n dalam deret tersebut diperoleh dengan menambahkan n ke istilah sebelumnya. Jadi, suku kedua ditemukan dengan menjumlahkan 2 untuk suku pertama (yaitu 1), memberikan 3. Suku ketiga adalah 3 ditambahkan ke suku kedua (yaitu 3) memberi 6, dan seterusnya.

Bilangan-bilangan dalam deretan ini disebut bilangan segitiga karena dapat divisualisasikan sebagai susunan objek segitiga, ditunjukkan sebagai kotak kecil pada Gambar 6.1.

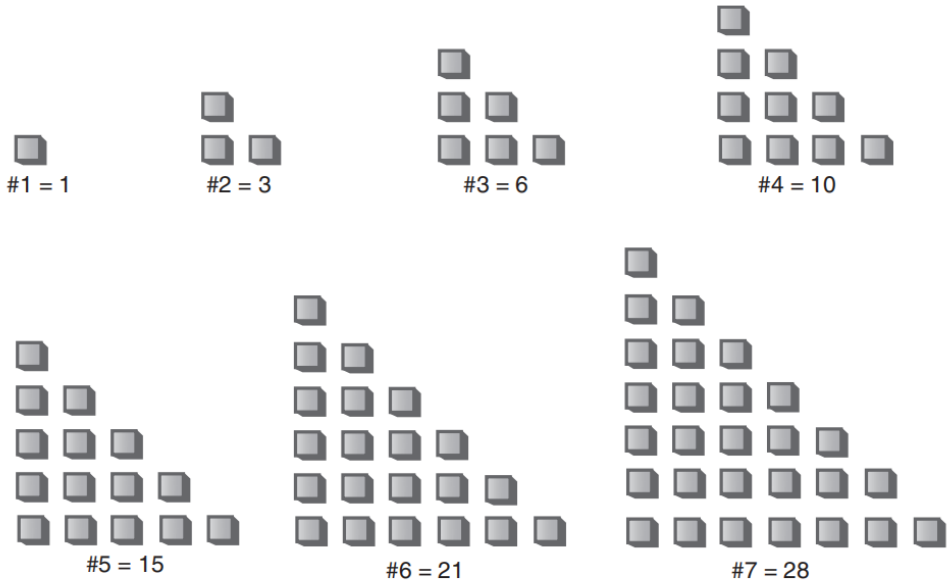

FIGURE 6.1 The triangular numbers.

Menemukan Suku ke-n Menggunakan Loop

Misalkan Anda ingin mencari nilai beberapa suku ke-n dalam rangkaian — misalnya suku keempat (yang nilainya 10). Bagaimana Anda menghitungnya? Melihat Gambar 6.2, Anda mungkin memutuskan bahwa nilai suku apa pun dapat diperoleh dengan menjumlahkan semuakol om vertikal kotak.

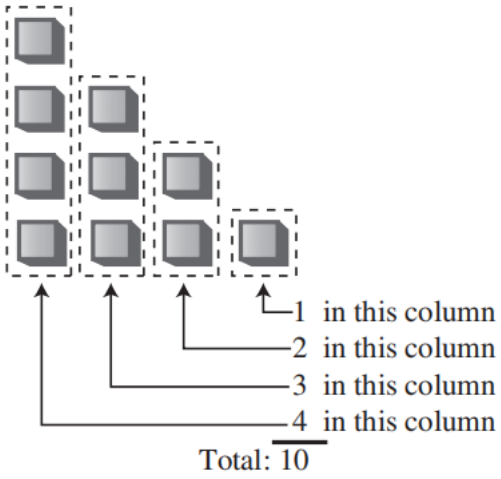

FIGURE 6.2 Triangular number as columns.

Dalam suku keempat, kolom pertama memiliki empat kotak kecil, kolom kedua memiliki tiga, dan seterusnya. Menambahkan 4 + 3 + 2 + 1 menghasilkan 10.

Metode segitiga () berikut menggunakan teknik berbasis kolom ini untuk mencari bilangan trian gular. Ini menjumlahkan semua kolom, dari ketinggian n hingga ketinggian 1:

```
int triangle(int n)
   {
   int total = 0;

   while(n > 0)           // until n is 1
      {
      total = total + n;  // add n (column height) to total
      --n;                // decrement column height
      }
   return total;
   }
```

Siklus metode di sekitar loop n kali, menambahkan n ke total pertama kali, n-1 the kedua kalinya, dan seterusnya turun ke 1, keluar dari loop ketika n menjadi 0.

Menemukan Istilah ke-n Menggunakan Rekursi

Pendekatan putaran mungkin tampak mudah, tetapi ada cara lain untuk melihatnya masalah. Nilai suku ke-n dapat dianggap sebagai penjumlahan dari hanya dua hal, bukan keseluruhan seri. Yaitu:

1. Kolom pertama (tertinggi), yang memiliki nilai n.
2. Jumlah dari semua kolom yang tersisa.

Ini ditunjukkan pada Gambar 6.3.

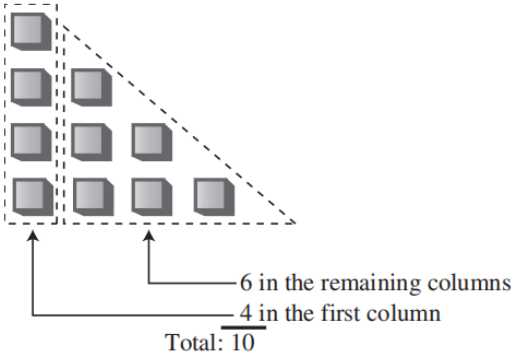

FIGURE 6.3 Triangular number as column plus triangle.

Menemukan Kolom Tersisa

Jika kita tahu tentang metode yang menemukan jumlah dari semua kolom yang tersisa, kita bisa menulis metode segitiga () kita, yang mengembalikan nilai segitiga ke-n nomor, seperti ini:

```
int triangle(int n)
   {
   return( n + sumRemainingColumns(n) );  // (incomplete version)
   }
```

Tapi apa yang kita peroleh di sini? Sepertinya menulis sumRemainingColumns ()metode ini sama sulitnya dengan menulis metode segitiga () di tempat pertama.

Perhatikan pada Gambar 6.3, bagaimanapun, bahwa jumlah dari semua kolom yang tersisa untuk istilah n adalah sama dengan jumlah dari semua kolom untuk term n-1. Jadi, jika kita mengetahui tentang a
metode yang menjumlahkan semua kolom untuk term n, kita bisa menyebutnya dengan argumen dari n-1 untuk mencari jumlah dari semua kolom yang tersisa untuk istilah n:

```
int triangle(int n)
   {
   return( n + sumAllColumns(n-1) );  // (incomplete version)
   }
```

Tetapi ketika Anda memikirkannya, metode sumAllColumns () melakukan hal yang persis sama Hal yang metode triangle () adalah: menjumlahkan semua kolom untuk beberapa nomor n berlalu sebagai argumen. Jadi mengapa tidak menggunakan metode segitiga () itu sendiri, alih-alih yang lain metode? Itu akan terlihat seperti ini:

```
int triangle(int n)
   {
   return( n + triangle(n-1) );  // (incomplete version)
   }
```

Anda mungkin heran bahwa suatu metode dapat memanggil dirinya sendiri, tetapi mengapa tidak dapat? SEBUAH pemanggilan metode adalah (antara lain) transfer kendali ke awal metode.Pengalihan kendali ini dapat terjadi dari dalam metode maupun dari luar.

Mengoper Buck

Semua pendekatan ini mungkin tampak seperti melewatkan tanggung jawab. Seseorang memberitahu saya untuk menemukan Angka segitiga ke-9. Saya tahu ini 9 ditambah bilangan segitiga ke-8, jadi saya panggil Harry dan memintanya untuk menemukan bilangan segitiga ke-8. Ketika saya mendengar kabar darinya, Saya akan menambahkan 9 untuk apa pun yang dia katakan kepada saya, dan itu akan menjadi jawabannya.

Harry tahu bilangan segitiga ke-8 sama dengan 8 ditambah bilangan segitiga ke-7, jadi dia memanggil Sally dan memintanya untuk menemukan nomor segitiga ke-7. Proses ini berlanjut dengan setiap orang memberikan tanggung jawab kepada satu sama lain.

Di manakah akhir pembagian uang ini? Seseorang pada suatu saat pasti bisa memahami jawaban yang tidak melibatkan meminta bantuan orang lain. Jika tidak terjadi, akan ada rantai orang yang tak terbatas yang menanyakan pertanyaan orang lain — a semacam skema Ponzi aritmatika yang tidak akan pernah berakhir. Dalam kasus segitiga (), ini berarti metode yang memanggil dirinya sendiri berulang-ulang dalam rangkaian tak terbatas yang akan akhirnya menghentikan program.

Buck Berhenti Di Sini

Untuk mencegah kemunduran tak terbatas, orang yang diminta untuk menemukan segitiga pertama nomor deret, jika n adalah 1, harus tahu, tanpa bertanya kepada orang lain, bahwa jawabannya adalah 1. Tidak ada angka yang lebih kecil untuk ditanyakan kepada siapa pun,

tidak ada yang tersisa untuk menambahkan ke hal lain, jadi tanggung jawab berhenti di situ. Kita dapat mengungkapkannya dengan menambahkan a
kondisi ke metode triangle ():

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

Kondisi yang mengarah ke metode rekursif kembali tanpa membuat yang lain panggilan rekursif disebut sebagai kasus dasar. Sangat penting bahwa setiap metode rekursif memiliki kasus dasar untuk mencegah rekursi tak terbatas dan akibat dari kematian program.

Program triangle.java

Apakah rekursi benar-benar berfungsi? Jika Anda menjalankan program triangle.java, Anda akan melihatnyatidak. Masukkan nilai untuk nomor term, n, dan program akan menampilkan nilaiangka segitiga yang sesuai. Kode 6.1 menunjukkan program triangle.java.

LISTING 6.1 The `triangle.java` Program

```
// triangle.java
// evaluates triangular numbers
// to run this program: C>java TriangleApp
import java.io.*;                // for I/O
class TriangleApp
   {
   static int theNumber;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      theNumber = getInt();
      int theAnswer = triangle(theNumber);
      System.out.println("Triangle="+theAnswer);
      }  // end main()
```

```
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // end class TriangleApp
```

Rutin main () meminta pengguna untuk nilai n, memanggil segitiga (), dan menampilkan nilai kembali. Metode segitiga () memanggil dirinya sendiri berulang kali untuk melakukan semua pekerjaan.

Berikut beberapa contoh keluarannya:
Masukkan angka: 1000
Segitiga = 500500

Kebetulan, jika Anda skeptis dengan hasil yang dikembalikan dari segitiga (), Anda bisa periksa dengan menggunakan rumus berikut:
bilangan segitiga ke-n = (n2+ n) / 2

Apa yang Sebenarnya Terjadi?
Mari kita ubah metode segitiga () untuk memberikan wawasan tentang apa yang terjadisaat dijalankan. Kami akan memasukkan beberapa pernyataan keluaran untuk melacak argumen dan mengembalikan nilai:

```
public static int triangle(int n)
   {
   System.out.println("Entering: n=" + n);
   if(n==1)
      {
      System.out.println("Returning 1");
      return 1;
      }
   else
      {
      int temp = n + triangle(n-1);
      System.out.println("Returning " + temp);
      return temp;
      }
   }
```

Berikut interaksi saat metode ini menggantikan segitiga sebelumnya () metode dan pengguna memasukkan 5:

```
Enter a number: 5

Entering: n=5
Entering: n=4
Entering: n=3
Entering: n=2
Entering: n=1
Returning 1
Returning 3
Returning 6
Returning 10
Returning 15

Triangle = 15
```

Setiap kali metode triangle () memanggil dirinya sendiri, argumennya, yang dimulai dari 5, adalah dikurangi 1. Metode itu terjun ke dalam dirinya sendiri lagi dan lagi sampai argumennya dikurangi menjadi 1. Kemudian metode itu kembali. Ini memicu serangkaian pengembalian. Itu metode naik kembali, seperti phoenix, dari versi yang dibuang itu sendiri. Setiap waktu itu mengembalikan, itu menambahkan nilai n yang dipanggil dengan ke nilai yang dikembalikan dari metode itu dipanggil.

Nilai yang dikembalikan merekapitulasi rangkaian bilangan segitiga, sampai jawabannya adalahkembali ke main (). Gambar 6.4 menunjukkan bagaimana setiap pemanggilan metode triangle () bisa dibayangkan sebagai "di dalam" yang sebelumnya.

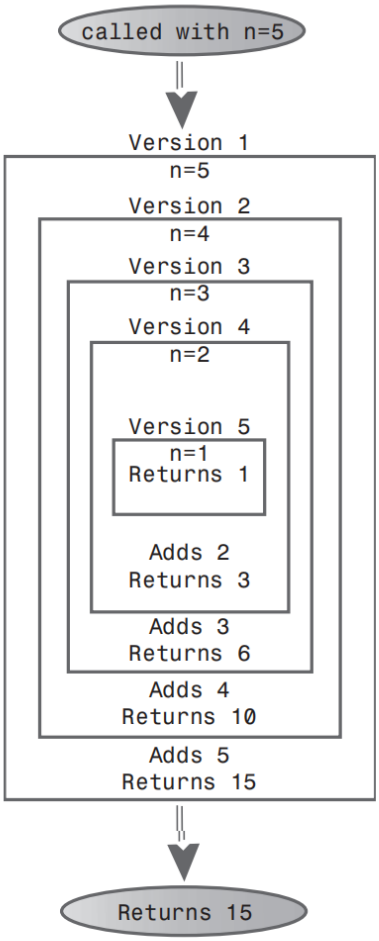

FIGURE 6.4 The recursive `triangle()` method.