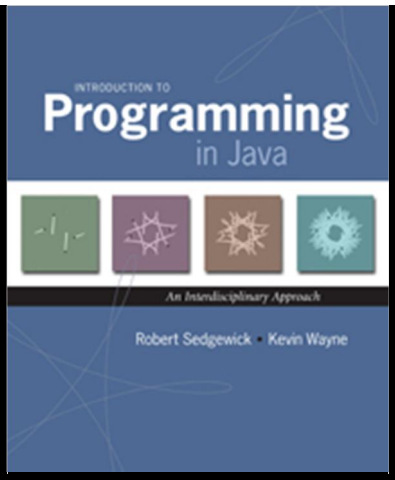
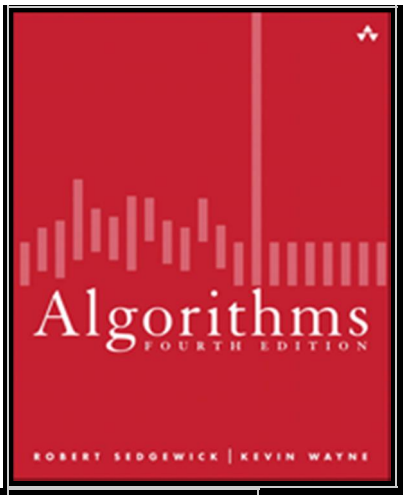


- **INTRO TO PROGRAMMING**
  - 1. Elements of Programming
    - 1.1 Your First Program1.2 Built-in Types of Data1.3 Conditionals and Loops1.4 Arrays1.5 Input and Output1.6 Case Study: PageRank2. Functions
    - 2.1 Static Methods2.2 Libraries and Clients2.3 Recursion2.4 Case Study: Percolation3. OOP
    - 3.1 Data Types3.2 Creating Data Types3.3 Designing Data Types3.4 Case Study: N-Body3.5 Purple America3.6 Inheritance4. Data Structures
    - 4.1 Performance4.2 Sorting and Searching4.3 Stacks and Queues4.4 Symbol Tables4.5 Case Study: Small World**INTRO TO CS**
  - 0. Prologue
  - 5. A Computing Machine
    - 5.1 Data Representations5.2 TOY Machine5.3 TOY Instruction Set5.4 TOY Programming5.5 TOY Simulator6. Building a Computer
    - 6.1 Combinational Circuits6.2 Sequential Circuits6.3 Building a TOY7. Theory of Computation
    - 7.1 Formal Languages7.2 Regular Expressions7.3 Finite State Automata7.4 Turing Machines7.5 Universality7.6 Computability7.7 Intractability7.8 Cryptography8. Systems
    - 8.1 Library Programming8.2 Compilers8.3 Operating Systems8.4 Networking8.5 Applications Systems9. Scientific Computation
    - 9.1 Floating Point9.2 Symbolic Methods9.3 Numerical Integration9.4 Differential Equations9.5 Linear Algebra9.6 Optimization9.7 Data Analysis9.8 Simulation**ALGORITHMS, 4TH EDITION**



2.3 RECURSION

The idea of calling one function from another immediately suggests the possibility of a function calling *itself*. The function-call mechanism in Java supports this possibility, which is known as *recursion*. Recursion is a powerful general-purpose programming technique, and is the key to numerous critically important computational applications, ranging from combinatorial search and sorting methods methods that provide basic support for information processing (Chapter 4) to the Fast Fourier Transform for signal processing (Chapter 9).

Your first recursive program.

The `HelloWorld` for recursion is to implement the *factorial* function, which is defined for positive integers N by the equation
N! = N × (N-1) × (N-2) × ... × 2 × 1
N! is easy to compute with a `for` loop, but an even easier method in Factorial.java is to use the following recursive function:

```
public static int factorial(int N) {
   if (N == 1) return 1;
   return N * factorial(N-1);
}
```

You can persuade yourself that it produces the desired result by noting that `factorial()` returns 1 = 1! when `N` is 1 and that if it properly computes the value
(N-1)! = (N-1) × (N-2) × ... × 2 × 1
then it properly computes the value
N! = N × (N-1)! = N × (N-1) × (N-2) × ... × 2 × 1
We can trace this computation in the same way that we trace any sequence of function calls.

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

Our `factorial()` implementation exhibits the two main components that are required for every recursive function.

- The *base case* returns a value without making any subsequent recursive calls. It does this for one or more special input values for which the function can be evaluated without recursion. For `factorial()`, the base case is N = 1.
- The *reduction step* is the central part of a recursive function. It relates the function at one (or more) inputs to the function evaluated at one (or more) other inputs. Furthermore, the sequence of parameter values must *converge* to the base case. For `factorial()`, the reduction step is `N * factorial(N-1)` and `N` decreases by one for each call, so the sequence of parameter values converges to the base case of `N = 1`.

Mathematical induction.

Recursive programming is directly related to *mathematical induction*, a technique for proving facts about discrete functions. Proving that a statement involving an integer *N* is true for infinitely many values of N by mathematical induction involves two steps.

- The *base case* is to prove the statement true for some specific value or values of *N* (usually 0 or 1).
- The *induction step* is the central part of the proof. For example, we typically assume that a statement is true for all positive integers less than *N*, then use that fact to prove it true for *N*.

Such a proof suffices to show that the statement is true for all *N*: we can start at the base case, and use our proof to establish that the statement is true for each larger value of *N*, one by one.

Euclid's algorithm.

The *greatest common divisor* (gcd) of two positive integers is the largest integer that divides evenly into both of them. For example, the greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68.

We can efficiently compute the gcd using the following property, which holds for positive integers *p* and *q*:

> If *p* > *q*, the gcd of *p* and *q* is the same as the gcd of *q* and *p* % *q*.

The static method `gcd()` in Euclid.java is a compact recursive function whose reduction step is based on this property.

```
gcd(1440, 408)
   gcd(408, 216)
      gcd(216, 192)
         gcd(192, 24)
            gcd(24, 0)
               return 24
            return 24
         return 24
      return 24
   return 24
```

Towers of Hanoi.

No discussion of recursion would be complete without the ancient *towers of Hanoi* problem. We have three poles and *n* discs that fit onto the poles. The discs differ in size and are initially arranged on one of the poles, in order from largest (disc *n*) at the bottom to smallest (disc 1) at the top. The task is to move the stack of discs to another pole, while obeying the following rules:

- Move only one disc at a time.
- Never place a disc on a smaller one.

To solve the problem, our goal is to issue a sequence of instructions for moving the discs. We assume that the poles are arranged in a row, and that each instruction to move a disc specifies its number and whether to move it left or right. If a disc is on the left pole, an instruction to move left means to wrap to the right pole; if a disc is on the right pole, an instruction to move right means to wrap to the left pole.

Exponential time.

One legend says that the world will end when a certain group of monks solve the Towers of Hanoi problem in a temple with 64 golden discs on three diamond needles. We can estimate the amount of time until the end of the world (assuming that the legend is true). If we define the function T(n) to be the number of move directives issued by TowersOfHanoi.java to move n discs from one peg to another, then the recursive code implies that T(n) must satisfy the following equation:
$T(n) = 2\ T(n - 1) + 1$ for $n > 1$, with $T(1) = 1$

Such an equation is known in discrete mathematics as a *recurrence relation*. We can often use them to derive a closed-form expression for the quantity of interest. For example, T(1) = 1, T(2) = 3, T(3) = 7, and T(4) = 15. In general, T(n) = 2^n - 1.

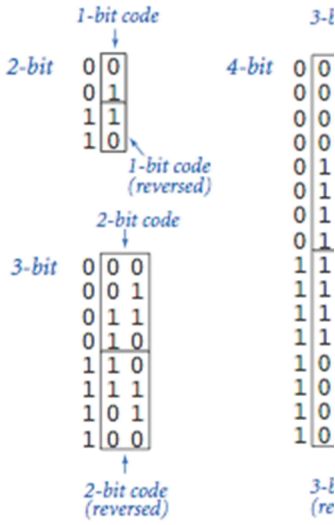
Knowing the value of T(n), we can estimate the amount of time required to perform all the moves. If the monks move discs at the rate of one per second, it would take more than one week for them to finish a 20-disc problem, more than 31 years to finish a 30-disc problem, and more than 348 centuries for them to finish a 40-disc problem (assuming that they do not make a mistake). The 64-disc problem would take more than 5.8 billion centuries.

Gray code.

The playwright Samuel Beckett wrote a play called *Quad* that had the following property: Starting with an empty stage, characters enter and exit one at a time, but each subset of characters on the stage appears exactly once. The play had four characters and there are 2^4 = 16 different subsets of four elements; hence the title. How did Beckett generate the stage directions for this play? How would we do it for 5 actors, or more?

code	subset	move
0 0 0 0	empty	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

Gray code representations



2-, 3-, and 4-bit Gray code

An *n*-bit *Gray code* is a list of the 2^*n* different *n*-bit binary numbers such that each entry in the list differs in precisely one bit from its predecessor. Gray codes directly apply to Beckett's problem because we can interpret each bit as denoting whether the integer corresponding to its bit position is in the subset. Changing the value of a bit from 0 to 1 corresponds to

an integer entering the subset; changing a bit from 1 to 0 corresponds to and integer exiting the subset.

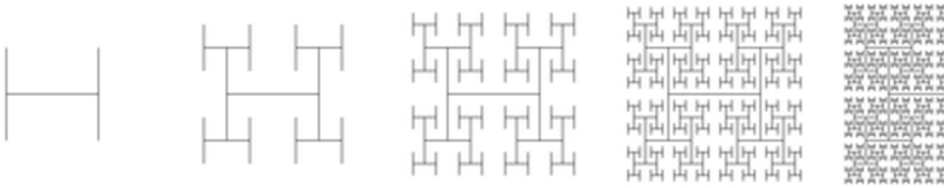
How do we generate a Gray code? A recursive plan that is very similar to the one that we used for the towers of Hanoi problem is effective. The *n* bit binary reflected Gray code is defined recursively as follows:

- the n-1 bit code, with 0 prepended to each word, followed by
- the n-1 bit code in reverse order, with 1 prepended to each word.

The 0-bit code is defined to be null, so the 1-bit code is 0 followed by 1. The recursive definition leads, after some careful thought, to the implementation in Beckett.java for printing out Beckett's stage directions.

Recursive graphics.

Simple recursive drawing schemes can lead to pictures that are remarkably intricate. For example, an *H-tree of order n* is defined as follows: The base case is null for n = 0. The reduction step is to draw, within the unit square three lines in the shape of the letter H four H-trees of order n-1, one connected to each tip of the H with the additional provisos that the H-trees of order n-1 are centered in the four quadrants of the square, halved in size. Program Htree.java takes a command-line argument n, and plots an order n H-tree using standard draw.



Brownian bridge.

An H-tree is a simple example of a *fractal*: a geometric shape that can be divided into parts, each of which is (approximately) a reduced size copy of the original. The study of fractals plays an important and lasting role in artistic expression, economic analysis, and scientific discovery. Artists and scientists use them to build compact models of complex shapes that arise in

nature and resist description using conventional geometry, such as clouds, plants, mountains, riverbeds, human skin, and many others. Economists also use fractals to model function graphs of economic indicators.

Program Brownian.java produces a function graph that approximates a simple example known as a *Brownian bridge* and closely related functions. You can think of this graph as a random walk that connects two points, from (x0, y0) to (x1, y1), controlled by a few parameters. The implementation is based on the *midpoint displacement method*, which is a recursive plan for drawing the plot within an interva [x0, x1]. The base case (when the size of the interval is smaller than a given tolerance) is to draw a straight line connecting the two endpoints. The reduction case is to divide the interval into two halves, proceeding as follows:

- Compute the midpoint (xm, ym) of the interval.
- Add to the *y*-coordinate of the midpoint a random value δ, chosen from the Gaussian distribution with mean 0 and given variance.
- Recur on the subintervals, dividing the variance by a given scaling factor *s*.

The shape of the curve is controlled by two parameters: the *volatility* (initial value of the variance) controls the distance the graph strays from the straight line connecting the points, and the *Hurst exponent* controls the smoothness of the curve. We denote the Hurst exponent by H and divide the variance by 2^(2H) at each recursive level. When H is 1/2 (divide by 2 at each level) the standard deviation is constant throughout the curve: in this situation, the curve is a Brownian bridge.

Pitfalls of recursion.

With recursion, you can write compact and elegant programs that fail spectacularly at runtime.

- *Missing base case.* The recursive function in NoBaseCase.java is supposed to compute Harmonic numbers, but is missing a base case:

```
public static double H(int N) {
   return H(N-1) + 1.0/N;
}
```

  If you call this function, it will repeatedly call itself and never return.

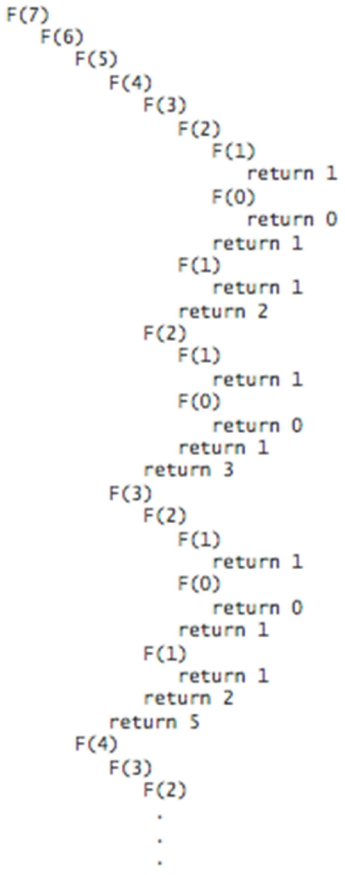
- *No guarantee of convergence.* Another common problem is to include within a recursive function a recursive call to solve a subproblem that is not smaller. The recursive function in NoConvergence.java will go into an infinite recursive loop if it is invoked with an argument N having any value other than 1:

```
public static double H(int N) {
   if (N == 1) return 1.0;
   return H(N) + 1.0/N;
}
```

- *Excessive space requirements.* Java needs to keep track of each recursive call to implement the function abstraction as expected. If a function calls itself recursively an excessive number of times before returning, the space required by Java for this task may be prohibitive. The recursive function in ExcessiveSpace.java correctly computes the Nth harmonic number. However, we cannot use it for large N because the recursive depth is proportional to N, and this creates a `StackOverflowError` for N = 5,000.

```
public static double H(int N) {
   if (N == 0) return 0.0;
   return H(N-1) + 1.0/N;
}
```

- *Excessive recomputation.* The temptation to write a simple recursive program to solve a problem must always be tempered by the understanding that a simple program might require exponential time (unnecessarily), due to excessive recomputation. For example, the Fibonacci sequence



Wrong way to compute Fibonacci numbers

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 ...

is defined by the formula $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$ with $F_0 = 0$ and $F_1 = 1$. A novice programmer might implement this recursive function to compute numbers in the Fibonacci sequence, as in Fibonacci.java:

```
public static long F(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return F(n-1) + F(n-2);
}
```

However, this program is spectacularly inefficient! To see why it is futile to do so,

consider what the function does to compute F(8) = 21. It first computes F(7) = 13 and F(6) = 8. To compute F(7), it recursively computes F(6) = 8 *again* and F(5) = 5. Things rapidly get worse because both times it computes F(6) it ignores the fact that it already computed F(5), and so forth. The the number of times this program computes F(1) when computing F(n) is precisely F_n. The mistake of recomputation is compounded, exponentially. No imaginable computer will ever be able to do this many calculations.

Q + A

Q. Are there situations when iteration is the only option available to address a problem?

A. No, any loop can be replaced by a recursive function, though the recursive version might require excessive memory.

Q. Are there situations when recursion is the only option available to address a problem?

A. No, any recursive function can be replaced by an iterative counterpart. In Section 4.3, we will see how compilers produce code for function calls by using a data structure called a *stack*.

Q. Which should I prefer, recursion or iteration?

A. Whichever leads to the simpler, more easily understood, or more efficient code.

Q. I get the concern about excessive space and excessive recomputation in recursive code. Anything else to be concerned about?

A. Be extremely wary of creating arrays in recursive code. The amount of space used can pile up very quickly, as can the amount of time required for memory management.

Exercises

1. What happens if you run `factorial()` with negative value of `n`? With a large value, say 35?
2. Write a recursive program that computes the value of ln(N!).
3. Give the sequence of integers printed by a call to `ex233(6)`:

```
public static void ex233(int n) {
    if (n <= 0) return;
    StdOut.println(n);
    ex233(n-2);
    ex233(n-3);
    StdOut.println(n);
}
```

4. Give the value of `ex234(6)`:

```
public static String ex234(int n) {
    if (n <= 0) return "";
    return ex234(n-3) + n + ex234(n-2) +
n;
}
```

5. Criticize the folowing recursive function:

```
public static String ex235(int n) {
    String s = ex233(n-3) + n + ex235(n-
2) + n;
    if (n <= 0) return "";
    return s;
}
```

6. Given four positive integers `a`, `b`, `c`, and `d`, explain what value is computed by `gcd(gcd(a, b), gcd(c, d))`.
7. Explain in terms of integers and divisors the effect of the following Euclid-like function.

```
public static boolean gcdlike(int p, int
q) {
   if (q == 0) return (p == 1);
   return gcdlike(q, p % q);
}
```

   Solution. Returns whether `p` and `q` are relatively prime.

8. Consider the following recursive function.

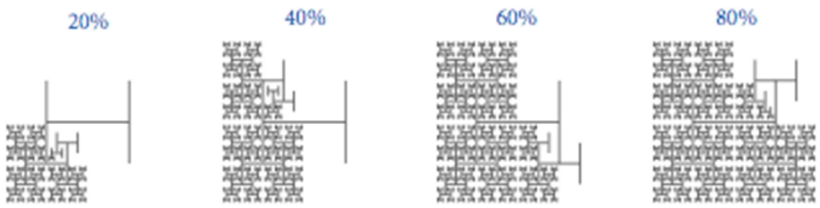
```
public static int mystery(int a, int b)
{
```

```
   if (b == 0)      return 0;
   if (b % 2 == 0) return mystery(a+a,
b/2);
   return mystery(a+a, b/2) + a;
}
```

What are the values of `mystery(2, 25)` and `mystery(3, 11)`? Given positive integers `a` and `b`, describe what value `mystery(a, b)` computes. Answer the same question, but replace `+` with `*` and replace `return 0` with `return 1`.

Solution. 50 and 33. It computes a*b. If you replace `+` with `*`, it computes a^b.

9. Write a recursive program `Ruler.java` to plot the subdivisions of a ruler using `StdDraw` as in Program 1.2.1.
10. Solve the following recurrence relations, all with T(1) = 1. Assume N is a power of two.
    - T(N) = T(N/2) + 1
    - T(N) = 2T(N/2) + 1
    - T(N) = 2T(N/2) + N
    - T(N) = 4T(N/2) + 3
11. Prove by induction that the minimum possible number of moves needed to solve the towers of Hanoi satisfies the same recurrence as the number of moves used by our recursive solution.
12. Prove by induction that the recursive program given in the text makes exactly F(n) recursive calls to F(1) when computing F(N).
13. Prove that the second argument to `gcd()` decreases by at least a factor of two for every second recursive call, then prove that gcd(p, q) uses at most $\log_2 N$ recursive calls, where N is the larger of p and q.
14. Write a program AnimatedHtree.java that animates the drawing of the H-tree.



Next, rearrange the order of the recursive calls (and the base case), view the resulting animation, and explain each outcome.

Creative Exercises

15. **Binary representation.** Write a program IntegerToBinary.java that takes a positive integer N (in decimal) from the command line and prints out its binary representation. Recall, in program 1.3.7, we used the method of subtracting out powers of 2. Instead, use the following simpler method: repeatedly divide 2 into N and read the remainders backwards. First, write a `while` loop to carry out this computation and print the bits in the wrong order. Then, use recursion to print the bits in the correct order.

16. **A4 paper.** The width-to-height ratio of paper in the ISO format is the square root of 2 to 1. Format A0 has an area of 1 square meter. Format A1 is A0 cut with a vertical line into two equal halves, A2 is A1 cut with a horizontal line into in two halves, and so on. Write a program that takes a command-line argument n and uses `StdDraw` to show how to cut a sheet of A0 paper into 2^n pieces. Here's a nice illustration of A size formats.

17. **Permutations.** Write a program Permutations.java that take a command-line argument n and prints out all n! permutations of the n letters starting at `a` (assume that n is no greater than 26). A *permutation* of n elements is one of the n! possible orderings of the elements. As an example, when n = 3 you should get the following output. Do not worry about the order in which you enumerate them.

```
bca cba cab acb bac abc
```

18. **Permutations of size k.** Modify Permutations.java to PermutationsK.java so that it takes two command-line arguments n and k, and prints out all P(n, k) = n! / (n-k)! permutations that contain exactly k of the n elements. Below is the desired output when k = 2 and n = 4. You need not print them out in any particular order.

```
ab ac ad ba bc bd ca cb cd da db dc
```

19. **Combinations.** Write a program Combinations.java that takes one command-line argument n and prints out all 2^n *combinations* of any size. A combination is a subset of the n elements, independent of order. As an example, when n = 3 you should get the following output.

```
 a ab abc ac b bc c
```

Note that the first element printed is the empty string (subset of size 0).

20. **Combinations of size k.** Modify Combinations.java to CombinationsK.java so that it takes two command-line arguments n and k, and prints out all C(n, k) = n! / (k! * (n-k)!) *combinations* of size k. For example, when n = 5 and k = 3 you should get the following output.

```
abc abd abe acd ace ade bcd bce bde cde
```

Alternate solution using arrays instead of strings: Comb2.java.

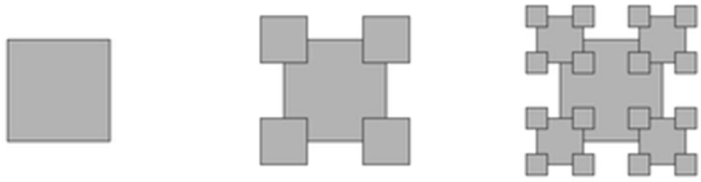
21. **Hamming distance.** The Hamming distance between two bit strings of length n is equal to the number of bits in which the two strings differ. Write a program that reads in an integer k and a bit string s from the command line, and prints out all bit strings that have Hamming distance at most k from s. For example if k is 2 and s is 0000 then your program should print out

```
0011 0101 0110 1001 1010 1100
```

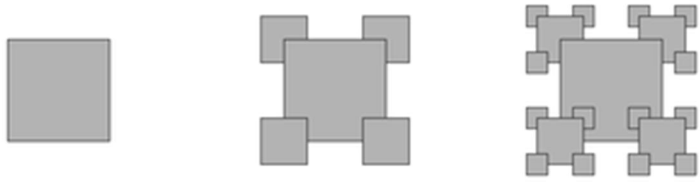
Hint: choose k of the N bits in s to flip.

22. **Recursive squares.** Write a program to produce each of the following recursive patterns. The ratio of the sizes of the squares is 2.2. To draw a shaded square, draw a filled gray square, then an unfilled black square.

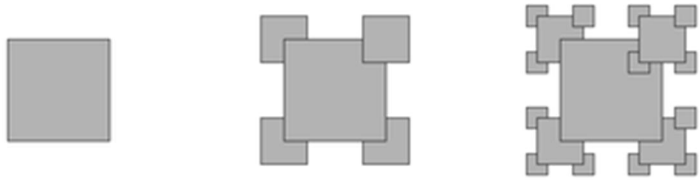
a.



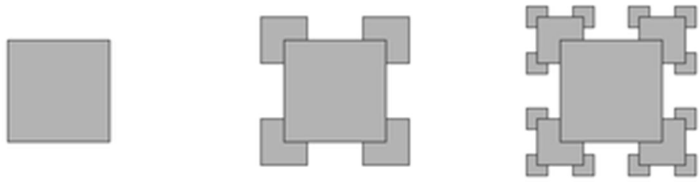
b.



c.



d.



23. RecursiveSquares.java gives a solution to part a.

24. **Pancake flipping.** You have a stack of N pancakes of varying sizes on a griddle. Your goal is to re-arrange the stack in descending order so that the largest pancake is on the bottom and the smallest one is on top. You are only permitted to flip the top k pancakes, thereby reversing their order. Devise a

scheme to arrange the pancakes in the proper order by using at most 2N - 3 flips. *Hint*: you can try out strategies here.

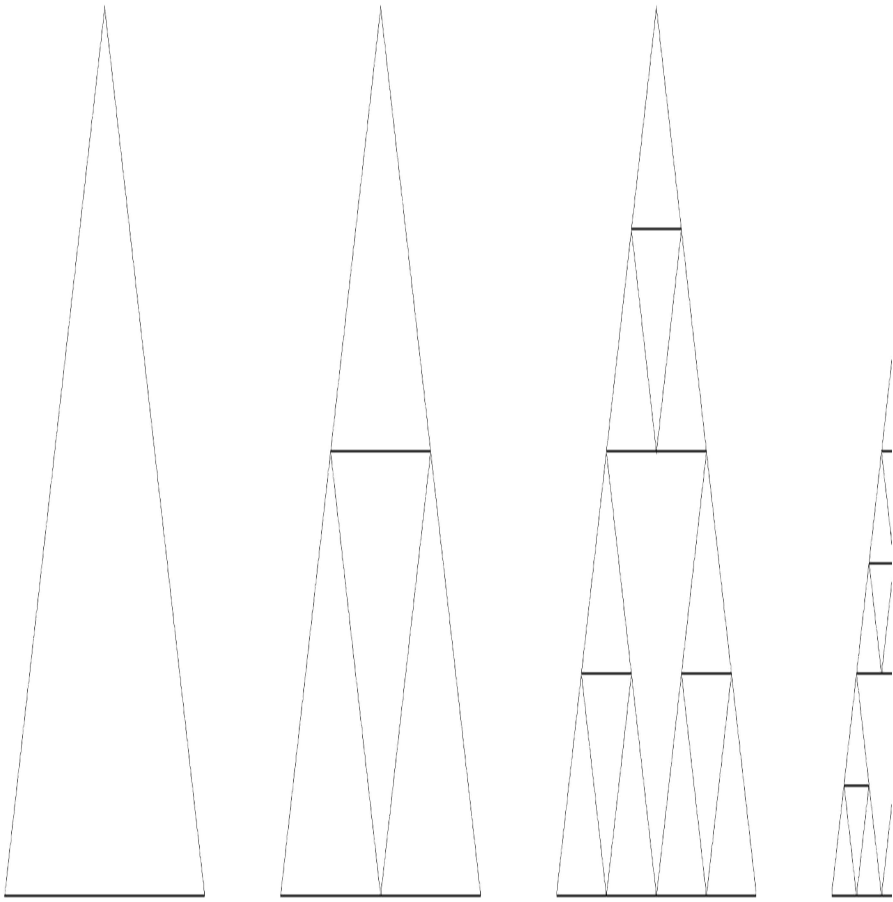
25. **Gray code.** Modify Beckett.java to print out the Gray code (not just the sequence of bit positions that change).

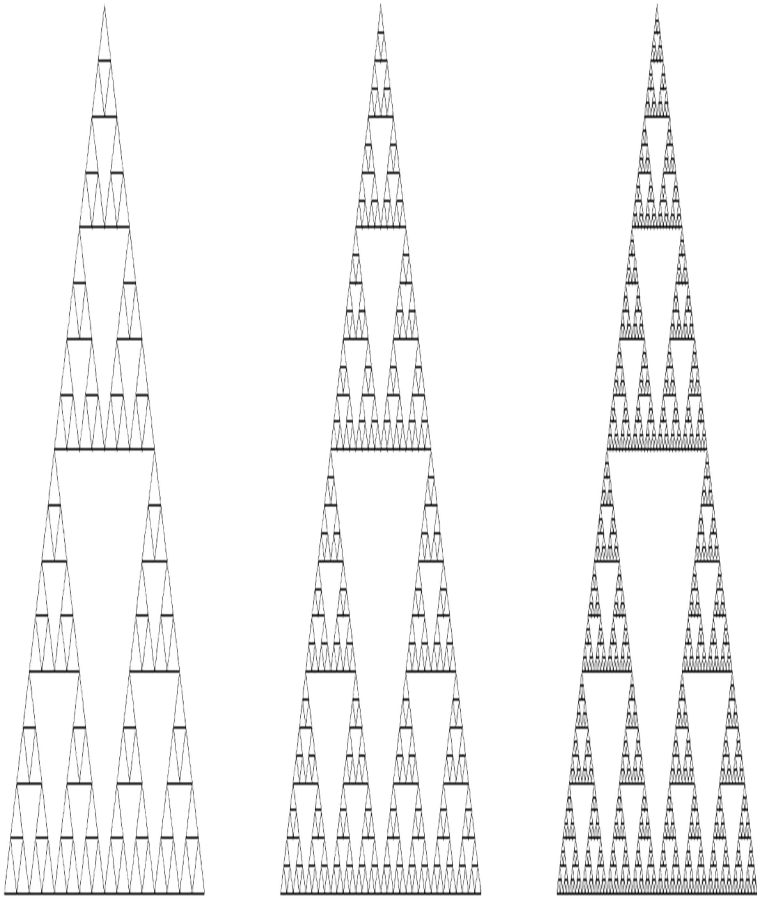
    Solution. GrayCode.java uses Java's string data type; GrayCodeArray.java uses a boolean array.

26. **Towers of Hanoi variant.** Consider the following variant of the towers of Hanoi problem. There are 2N discs of increasing size stored on three poles. Initially all of the discs with odd size (1, 3, ..., 2N-1) are piled on the left pole from top to bottom in increasing order of size; all of the discs with even size (2, 4, ..., 2N) are piled on the right pole. Write a program to provide instructions for moving the odd discs to the right pole and the even discs to the left pole, obeying the same rules as for towers of Hanoi.

27. **Animated towers of Hanoi animation.** Write a program AnimatedHanoi.java that uses `StdDraw` to animate a solution to the towers of Hanoi problem, moving the discs at a rate of approximately 1 per second.

28. **Sierpinski triangles.** Write a recursive program to draw the *Sierpinski gasket*. As with `Htree`, use a command-line argument to control the depth of recursion.





30. **Binomial distribution.** Estimate the number of recursive calls that would be used by the code

```
public static double binomial(int N, int
k) {
   if ((N == 0) || (k < 0)) return 1.0;
   return (binomial(N-1, k) +
binomial(N-1, k-1)) / 2.0;
}
```

to compute `binomial(100, 50)`. Develop a better implementation that is based on memoization. Hint: See Exercise 1.4.26.

31. **Collatz function.** Consider the following recursive function in Collatz.java, which is related to a famous unsolved problem in

number theory, known as the Collatz problem or the *3n + 1 problem*.

```
public static void collatz(int n) {
   StdOut.print(n + " ");
   if (n == 1) return;
   if (n % 2 == 0) collatz(n / 2);
   else            collatz(3*n + 1);
}
```

For example, a call to `collatz(7)` prints the sequence

> 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

as a consequence of 17 recursive calls. Write a program that takes a command-line argument N and returns the value of n < N for which the number of recursive calls for collatz(n) is maximized. Hint: use memoization. The unsolved problem is that no one knows whether the function terminates for all positive values of n (mathematical induction is no help because one of the recursive calls is for a larger value of the argument).

32. **Brownian island.** Benoit Mandelbrot asked the famous question *How long is the coast of Britain?* Modify Brownian.java to get a program BrownianIsland.java that plots Brownian islands, whose coastlines resemble that of Great Britain. The modifications are simple: first, change `curve()` to add a gaussian to the x coordinate as well as to the y coordinate; second, change `main()` to draw a curve from the point at the center of the canvas back to itself. Experiment with various values of the arguments to get your program to produce islands with a realistic look.

Brownian islands with Hurst exponent of .76

33. **Plasma clouds.** Write a recursive program Program PlasmaCloud.java to draw plasma clouds, using the method suggested in the text. *Remark*: this technique can be used to produce synthetic terrain, by interpreting the color value as the altitude. Variants of this scheme are widely used in the entertainment industry to generate artificial background scenery for movies and the "Genesis effect" in Star Trek II and the "outline of the "Death Star" in Return of the Jedi.

34. **A strange function.** Consider McCarthy's 91 function:

```
public static int mcCarthy(int n) {
    if (n > 100) return n - 10;
    else return
mcCarthy(mcCarthy(n+11));
 }
```

Determine the value of `mcCarthy(50)` without using a computer. Give the number of recursive calls used by `mcCarthy()` to compute this result. Prove that the base case is reached for all positive integers `n` or give a value of `n` for which this function goes into a recursive loop.

35. **Recursive tree.** Write a program Tree.java that takes a command-line arguemnt `N` and produces the following recurisve patterns for `N` equal to 1, 2, 3, 4, and 8.

Web Exercises

1. Does Euclid.java still work if the inputs can be negative? If not, fix it. *Hint*: Recall that % can return a negative integer if the first input is negative. When calling the function, take the absolute value of both inputs.
2. Write a recursive program GoldenRatio.java that takes an integer input N and computes an

approximation to the golden ratio using the following recursive formula:

```
f(N) = 1                if N = 0
     = 1 + 1 / f(N-1)   if N > 0
```

Redo, but do not use recursion.

3. Discover a connection between the golden ratio and Fibonacci numbers. *Hint*: consider the ratio of successive Fibonacci numbers: 2/1, 3/2, 8/5, 13/8, 21/13, 34/21, 55/34, 89/55, 144/89, ...
4. Consider the following recursive function. What is `mystery(1, 7)`?

```
public static int mystery(int a, int b)
{
   if (0 == b) return 0;
   else return a + mystery(a, b-1);
}
```

Will the function in the previous exercise terminate for every pair of integers a and b between between 0 and 100? Give a high level description of what `mystery(a, b)` returns, given integers a and b between 0 and 100.

Answer: mystery(1, 7) = 1 + mystery(1, 6) = 1 + (1 + mystery(1, 5)) = ... 7 + mystery(1, 0) = 7.

Answer: Yes, the base case is b = 0. Successive recursive calls reduce b by 1, driving it toward the base case. The function `mystery(a, b)` returns `a * b`. Mathematically inclined students can prove this fact via induction using the identity ab = a + a(b-1).

5. Consider the following function. What does `mystery(0, 8)` do?

```
public static void mystery(int a, int b)
{
   if (a != b) {
       int m = (a + b) / 2;
       mystery(a, m);
       StdOut.println(m);
       mystery(m, b);
   }
}
```

Answer: infinite loop.

6. Consider the following function. What does `mystery(0, 8)` do?

```
public static void mystery(int a, int b)
{
   if (a != b) {
       int m = (a + b) / 2;
       mystery(a, m - 1);
       StdOut.println(m);
       mystery(m + 1, b);
   }
}
```

Answer: stack overflow.

7. Repeat the previous exercise, but replace `if (a != b)` with `if (a <= b)`.
8. What does `mystery(0, 8)` do?

```
public static int mystery(int a, int b)
{
    if (a == b) StdOut.println(a);
    else {
       int m1 = (a + b    ) / 2;
       int m2 = (a + b + 1) / 2;
       mystery(a, m1);
       mystery(m2, b);
    }
}
```

9. What does the following function compute?

```
public static int f(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   if (n == 2) return 1;
   return 2*f(n-2) + f(n-3);
```

10. Write a program Fibonacci2.java that takes a command-line argument N and prints out the first N Fibonacci numbers using the following alternate definition:

```
F(n)   = 1                                  if
n = 1 or n = 2
       = F((n+1)/2)^2 + F((n-1)/2)^2        if
n is odd
       = F(n/2 + 1)^2 - F(n/2 - 1)^2        if
n is even
```

What is the biggest Fibonacci number you can compute in under a minute using this definition? Compare this to Fibonacci.java.

11. Write a program that takes a command-line argument N and prints out the first N Fibonacci numbers using the following method proposed by Dijkstra:

```
F(0) =  0
F(1) =  1
F(2n-1) = F(n-1)^2 + F(n)^2
F(2n) = (2F(n-1) + F(n)) * F(n)
```

12. Prove by mathematical induction that the alternate definitions of the Fibonacci function given in the previous two exercises are equivalent to the original definition.
13. Write a program `Pell.java` that takes a command-line argument N and prints out the first N *Pell numbers*: $p_0 = 0$, $p_1 = 1$, and for n >= 2, $p_n = 2\ p_{n-1} + p_{n-2}$. Print out the ratio of successive terms and compare to 1 + sqrt(2).
14. Consider the following function from program Recursion.java:

```
public static void mystery(int n) {
   if (n == 0 || n == 1) return;
   mystery(n-2);
   StdOut.println(n);
   mystery(n-1);
}
```

    What does `mystery(6)` print out? *Hint*: first figure out what `mystery(2)`, `mystery(3)`, and so forth print out.
15. What would happen in the previous exercise if the base case was replaced with the following statement?

```
if (n == 0) return;
```

16. Consider the following recursive functions.

```
public static int square(int n) {
   if (n == 0) return 0;
   return square(n-1) + 2*n - 1;
}

public static int cube(int n) {
   if (n == 0) return 0;
   return cube(n-1) + 3*(square(n)) -
3*n + 1;
}
```

What is the value of `square(5)`? `cube(5)`? `cube(123)`?

17. Consider the following pair of mutually recursive functions. What does `g(g(2))` evaluate to?

```
public static int f(int n) {
   if (n == 0) return 0;
   return f(n-1) + g(n-1);
}

public static int g(int n) {
   if (n == 0) return 0;
   return g(n-1) + f(n);
}
```

18. Write program to verify that (for small values of n) the sum of the cubes of the first n Fibonacci numbers F(0)^3 + F(1)^3 + ... + F(n)^3 equals (F(3n+4) + (-1)^n * 6 * f(n-1)) / 10, where F(0) = 1, F(1) = 1, F(2) = 2, and so forth.

19. **Transformations by increment and unfolding.** Given two integers a ≤ b, write a program Sequence.java that transforms a into b by a minimum sequence of increment (add 1) and unfolding (multiply by 2) operations. For example,

```
% java Sequence 5 23
23 = ((5 * 2 + 1) * 2 + 1)

% java Sequence 11 113
113 = ((((11 + 1) + 1) + 1) * 2 * 2 * 2
+ 1)
```

20. **Hadamard matrix.** Write a recursive program Hadamard.java that takes a command-line argument n and plots an N-by-N Hadamard pattern where $N = 2^n$. Do *not* use an array. A 1-by-1 Hadamard pattern is a single black square. In general a 2N-by-2N Hadamard pattern is obtained by aligning 4 copies of the N-by-N pattern in the form of a 2-by-2 grid, and then inverting the colors of all the squares in the lower right N-by-N copy. The N-by-N Hadamard H(N) matrix is a boolean matrix with the remarkable property that any two rows differ in exactly N/2 bits. This property makes it useful for designing *error-correcting codes*. Here are the first few Hadamard matrices.



21. **8 queens problem.** In this exercise, you will solve the classic 8-queens problem: place 8 queens on an 8x8 chess board so that no two queens are in the same row, column, or diagonal. There are 8! = 40,320 ways in which no two queens are placed in the same row or column. Any permutation p[] of the integers 0 to 7 gives such a placement: put queen i in row i, column p[i]. Your program Queens.java should take one command-line argument N and enumerate all solutions to the N-queens problem by drawing the location of the queens in ASCII like the two solutions below.

```
* * * Q * * * *          * * * * Q * * *
* Q * * * * * *          * Q * * * * * *
* * * * * * Q *          * * * Q * * * *
* * Q * * * * *          * * * * * * Q *
* * * * * Q * *          * * Q * * * * *
* * * * * * * Q          * * * * * * * Q
* * * * Q * * *          * * * * * Q * *
Q * * * * * * *          Q * * * * * * *
```

Hint: to determine whether setting q[n] = i conflicts with q[0] through q[n-1]

- if q[i] equals q[n]: two queens are placed in the same column
- if q[i] - q[n] equals n - i: two queens are on same major diagonal
- if q[n] - q[i] equals n - i: two queens are on same minor diagonal

22. **Another 8 queens solver.** Program Queens2.java solves the 8 queens problem by implicitly enumeration all N! permutations (instead of the N^N placements). It is based on program Permutations.java.
23. **Euclid's algorithm and π.** The probability that two numbers chosen from a large random set of numbers have no common factors (other than 1) is $6 / \pi^2$. Use this idea to estimate π.

Robert Matthews use the same idea to estimate π by taken the set of numbers to be a function of the positions of stars in the sky.

24. **Towers of Hanoi variant II.** (Knuth-Graham and Pathashnik) Solve the original Towers of Hanoi problem, but with the extra restriction that you are not allowed to directly transfer a disk from A to C. How many moves does it take to solve a problem with N disks? *Hint*: move N-1 smallest disks from A to C recursively (without any direct A to C moves), move disk N from A to B, move N-1 smallest disks from C to A (without any direct A to C moves), move disk N from B to C, and move N-1 smallest disks from A to C recursively (without any direct A to C moves).
25. **Towers of Hanoi variant III.** Repeat the previous question but disallow both A to C and C to A moves. That is, each move must involve pole B.
26. **Towers of Hanoi with 4 pegs.** Suppose that you have a fourth peg. What is the least number of moves needed to transfer a stack of 8 disks from the leftmost peg to the rightmost peg? Answer. Finding the shortest such solution in general has remained an open problem for over a hundred years and is known as *Reve's puzzle*.
27. **Another tricky recursive function.** Consider the following recursive function. What is `f(0)`?

```
public static int f(int x) {
   if (x > 1000) return x - 4;
   else return f(f(x+5));
}
```

28. **Checking if N is a Fibonacci number.** Write a function to check if N is a Fibonacci number. *Hint*: a positive integer is a Fibonacci number if and only if either (5*N*N + 4) or (5*N*N - 4) is a perfect square.
29. **Random infix expression generator.** Run Expr.java with different command-line argument p between 0 and 1. What do you observe?

```
public static String expr(double p) {
```

```
   double r = Math.random();
   if (r <= 1*p) return "(" + expr(p) +
" + " + expr(p) + ")";
   if (r <= 2*p) return "(" + expr(p) +
" * " + expr(p) + ")";
   return "" + (int) (100 *
Math.random());
}
```

30. **A tricky recurrence.** Define F(n) so that F(0) = 0 and F(n) = n - F(F(n-1)). What is F(100000000)?

    Solution: The answer is related to the Fibonacci sequence and the Zeckendorf representation of a number.

31. **von Neumann ordinal.** The *von Neumann integer* i is defined as follows: for i = 0, it is the empty set; for i > 0, it is the set containing the von Neumann integers 0 to i-1.

```
0 = {}          = {}
1 = {0}         = {{}}
2 = {0, 1}      = {{}, {{}}}
3 = {0, 1, 2}   = {{}, {{}}, {{}, {{}}}}
```

    Write a program Ordinal.java with a recursive function `vonNeumann()` that takes a nonnegative integer `N` and returns a string representation of the von Neumann integer N. This is a method for defining ordinals in set theory.

32. **Subsequences of a string.** Write a program Subsequence.java that takes a string command-line argument `s` and an integer command-line argument `k` and prints out all subsequences of `s` of length `k`.

```
% java Subsequence abcd 3
abc abd acd bcd
```

33. **Interleaving two strings.** Given two strings `s` and `t` of distinct characters, print out all (M+N)! / (M! N!) interleavings, where M and N are the number of characters in the two strings. For example, if

```
s = "ab"  t = "CD"
abCD   CabD
aCbD   CaDb
```

```
aCDb    CDab
```

34. **Binary GCD.** Write a program BinaryGCD.java that finds the greatest common divisor of two positive integers using the binary gcd algorithm: gcd(p, q) =
    - p if q = 0
    - q if p = 0
    - 2 * gcd(p/2, q/2) if p and q are even
    - gcd(p/2, q) if p is even and q is odd
    - gcd(p, q/2) if p is odd and q is even
    - gcd((p-q)/2, q) if p and q are odd and p >= q
    - gcd(p, (q-p)/2) if p and q are odd and p < q
35. **Integer partitions.** Write a program Partition.java that takes a positive integer N as a command-line argument and prints out all partitions of N. A partition of N is a way to write N as a sum of positive integers. Two sums are considered the same if they only differ in the order of their constituent summands. Partitions arise in symmetric polynomials and group representation theory in mathematics and physics.

```
% java Partition 4
6
4
3 1
2 2
2 1 1
1 1 1 1
```

```
% java Partition
6
5 1
4 2
4 1 1
3 3
3 2 1
3 1 1 1
2 2 2
2 2 1 1
2 1 1 1 1
1 1 1 1 1 1
```

36. **Johnson-Trotter permutations.** Write a program JohnsonTrotter.java that take a command-line argument n and prints out all n! permutations of the integer 0 through n-1 in such a way that consecutive permutations differ in only one adjacent transposition (similar to way Gray code iterates over combinations in

such a way that consecutive combinations differ in only one bit).

```
% java JohnsonTrotter 3
012   (2 1)
021   (1 0)
201   (2 1)
210   (0 1)
120   (1 2)
102   (0 1)
```

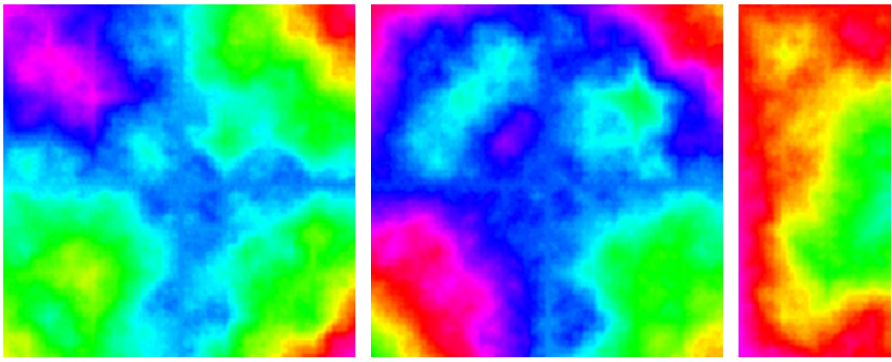
37. **Permutations in lexicographic order.** Write a program PermutationsLex.java that take a command-line argument N and prints out all N! permutations of the integer 0 through N-1 in lexicographic order.

```
% java PermutationsLex 3
012
021
102
120
201
210
```

38. **Derangements.** A derangement is a permutation `p[]` of the integers from 0 to N-1 such that p[i] doesn't equal i for any i. For example there are 9 derangmenets when N = 4: 1032, 1230, 1302, 2031, 2301, 2310, 3012, 3201, 3210. Write a program to count the number of derangements of size N using the following recurrence: d[N] = (N-1) (d[N-1] + d[N-2]), where d[1] = 0, d[2] = 1. The first few terms are 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, and 1334961.
39. **Tribonacci numbers.** The *tribonacci numbers* are similar to the Fibonacci numbers, except that each term is the sum of the three previous terms in the sequence. The first few terms are 0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81. Write a program to compute tribonacci numbers. What is the ratio successive terms? *Answer*. Root of x^3 - x^2 - x - 1, which is approximately 1.83929.
40. **Sum of first n Fibonacci numbers.** Prove by induction that the sum of the first n Fibonacci numbers F(1) + F(2) + ... + F(N) is F(N+2) - 1.
41. **Combinational Gray Code.** Print out all combination of k of n items in such a way that

consecutive combinations differ in exactly one element, e.g., if k = 3 and n = 5, 123, 134, 234, 124, 145, 245, 345, 135, 235, 125. *Hint*: use the Gray code, but only print out those integers with exactly k 1's in their binary representation.

42. **Maze generation.** Create a maze using divide-and-conquer: Begin with a rectangular region with no walls. Choose a random gridpoint in the rectangle and construct two perpendicular walls, dividing the square into 4 subregions. Choose 3 of the four regions at random and open a one cell hole at a random point in each of the 3. Recur until each subregion has width or height 1.
43. **Plasma clouds.** Program PlasmaCloud.java takes a command-line argument N and produces a random N-by-N plasma fractal using the midpoint displacement method.



44. Here's an 800-by-800 example. Here's a reference, including a simple 1D version. Note: some visual artifacts are noticeable parallel to the x and y axes. Doesn't have all of the statistical properties of 2D fractional Brownian motion.
45. **Fern fractal.** Write a recursive program to draw a fern or tree, as in this fern fractal demo.
46. **Integer set partition.** Use memoization to develop a program that solves the set partition problem for positive integer values. You may use an array whose size is the sum of the input values.
47. **Voting power.** John F. Banzhaf III proposed a ranking system for each coalition in a block voting system. Suppose party i control w[i]

votes. A strict majority of the votes is needed to accept or reject a proposal. The voting power of party i is the number of minority coalitions it can join and turn it into a winning majority coalition. Write a program VotingPower.java that takes in a list of coalition weights as command-line argument and prints out the voting power of each coalition. *Hint*: use Schedule.java as a starting point.

48. **Scheduling on two parallel machines.** Program Schedule.java takes a command-line argument N, reads in N real number of standard input, and partitions them into two groups so that their difference is minimized.

Last modified on March 02, 2012.