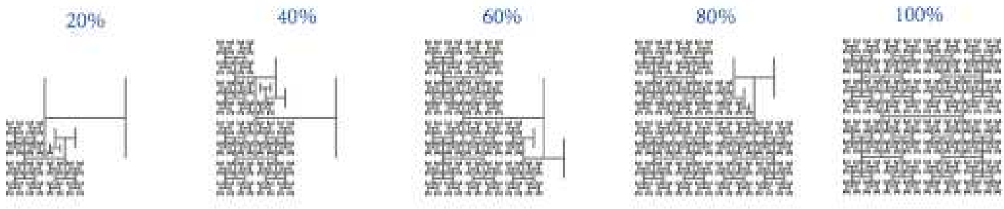
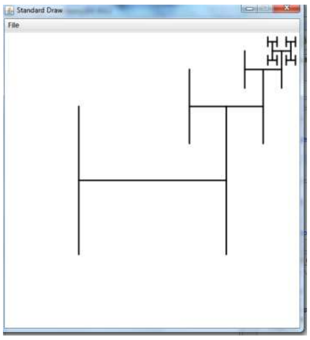
Section 4

1. 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.

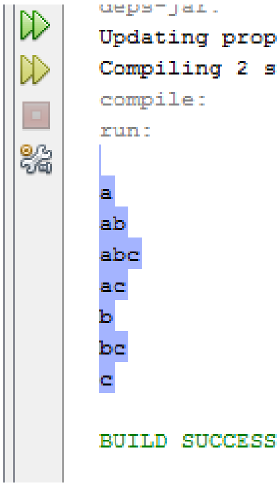


The first H starts to appear in a different place, depending in which is placed.

2. **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.

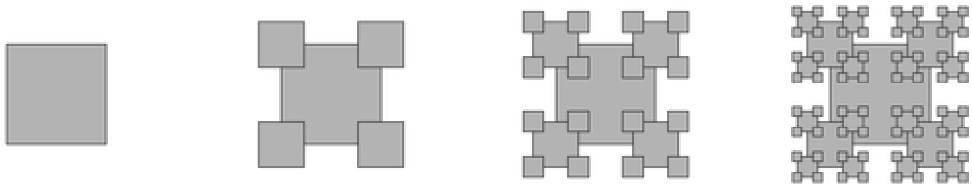
```
 a ab abc ac b bc c
```

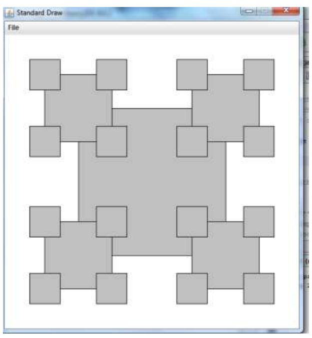
Note that the first element printed is the empty string (subset of size 0).



3. **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.





// recursively draw 4 smaller trees of order n-1

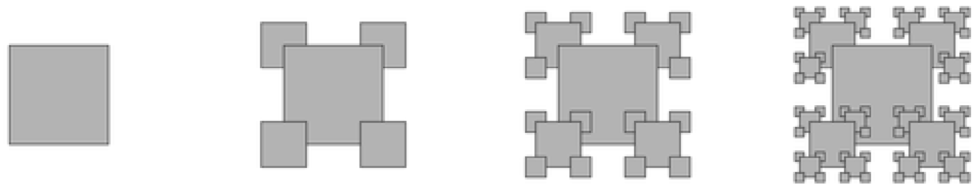
draw(n-1, x - size/2, y - size/2, size/ratio);   // lower left

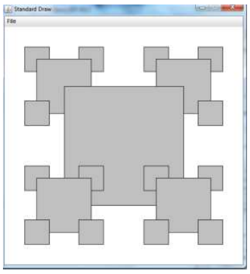
draw(n-1, x - size/2, y + size/2, size/ratio);   // upper left

draw(n-1, x + size/2, y - size/2, size/ratio);   // lower right

```
draw(n-1, x + size/2, y + size/2, size/ratio);    // upper right
```

b.

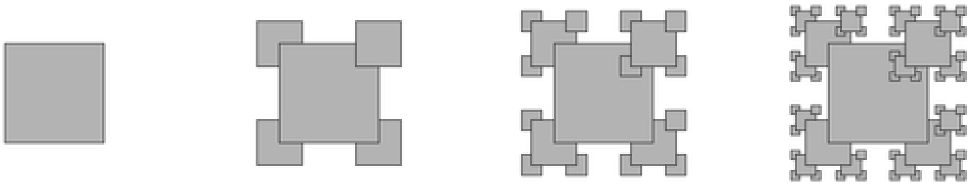


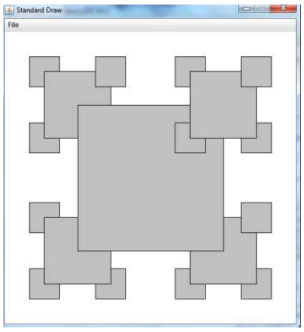


```
// recursively draw 4 smaller trees of order n-1

draw(n-1, x + size/2, y + size/2, size/ratio);    // upper right
draw(n-1, x - size/2, y + size/2, size/ratio);    // upper left

drawSquare(x, y, size);

draw(n-1, x - size/2, y - size/2, size/ratio);    // lower left
draw(n-1, x + size/2, y - size/2, size/ratio);    // lower right
```

Section 4

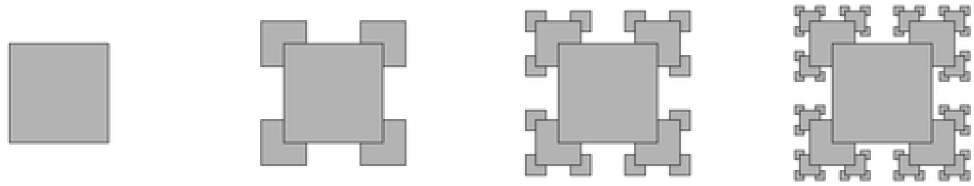
c.



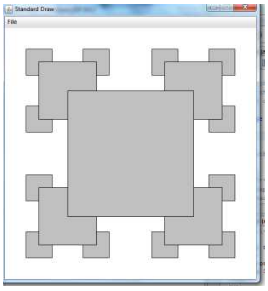



```
// recursively draw 4 smaller trees of order n-1

        draw(n-1, x - size/2, y + size/2, size/ratio);    // upper left

        draw(n-1, x - size/2, y - size/2, size/ratio);    // lower left

  draw(n-1, x + size/2, y - size/2, size/ratio);    // lower right


    drawSquare(x, y, size);

 draw(n-1, x + size/2, y + size/2, size/ratio);    // upper right
```

d.



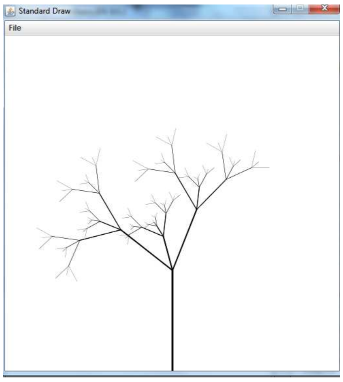
RecursiveSquares.java gives a solution to part a.



// recursively draw 4 smaller trees of order n-1

```
    draw(n-1, x - size/2, y - size/2, size/ratio);    // lower left
    draw(n-1, x - size/2, y + size/2, size/ratio);    // upper left
    draw(n-1, x + size/2, y - size/2, size/ratio);    // lower right
    draw(n-1, x + size/2, y + size/2, size/ratio);    // upper right


        drawSquare(x, y, size);
```

4. **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.



5. 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.

6. 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:

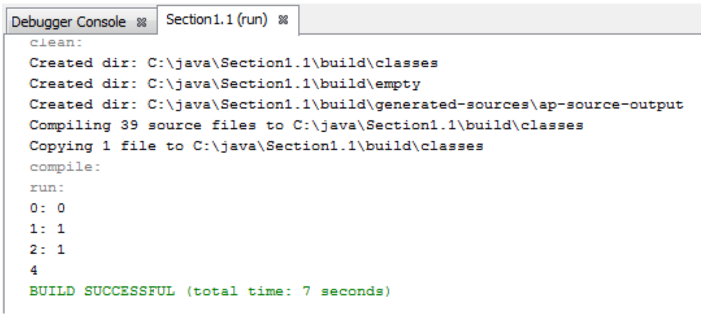
$$
\begin{aligned}
F(n) &= 1 && \text{if } n = 1 \text{ or } n = 2 \\
&= F((n+1)/2)^2 + F((n-1)/2)^2 && \text{if } n \text{ is odd} \\
&= F(n/2 + 1)^2 - F(n/2 - 1)^2 && \text{if } n \text{ is even}
\end{aligned}
$$

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

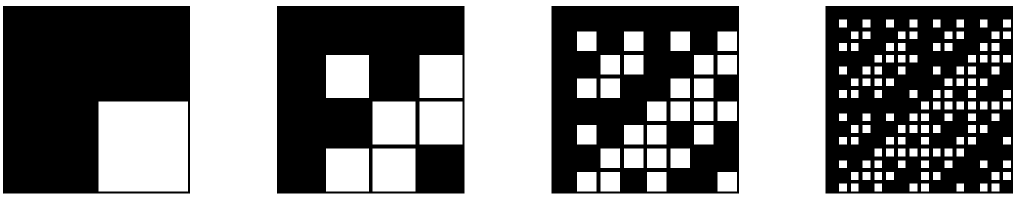
```
39: 63245986
40: 102334155
41: 165580141
42: 267914296
43: 433494437
44: 701408733
BUILD SUCCESSFUL (total time: 1 minute 7 seconds)
```

7. 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:

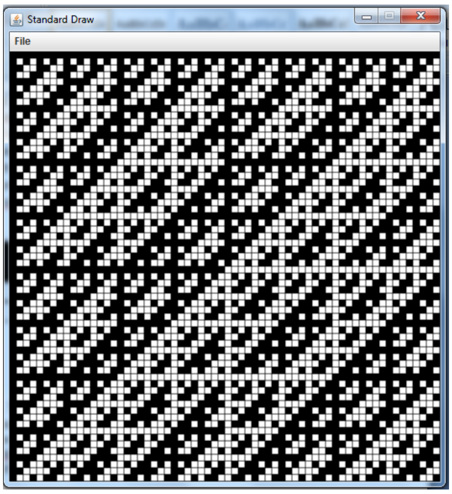
```
F(0) =  0
F(1) =  1
F(2n-1) = F(n-1)^2 + F(n)^2
F(2n) = (2F(n-1) + F(n)) * F(n)
```



8. **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.



Section 4



9. **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.

```
32: 53798080
33: 98950096
34: 181997601
35: 334745777
36: 615693474
37: 1132436852
BUILD SUCCESSFUL (total time: 1 minute 10 seconds)
```

10. **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.