Question 1

```
Fib(n):
    i = 1
    fib1 = 1
    fib2 = 1
    temp = 0

    while i < n  do
        temp = fib2
        fib2 = fib1 + fib2
        fib1 = temp
        i = i + 1

    return fib2
```

This algorithm takes as input any nonnegative integer *n*, and returns the *n*th Fibonacci number.

Explanation:

In the beginning of the *while* loop, `fib2` represents `Fib(i)`, and `fib1` represents `Fib(i-1)`. If `n` is 0 or 1, then the *while* loop is not run at all, and 1 (the correct result) is returned. Otherwise, in each iteration, the algorithm effectively puts the sum of `fib2` and `fib1` (which sum is equal to `Fib(i+1)`) in `fib2`, and puts the previous value of `fib2` (which is equal to `Fib(i)`) in `fib1`. Then, `i` increases by 1, so `fib2` again corresponds to `Fib(i)`, and `fib1` to `Fib(i-1)`.

The loop will end when `i` reaches `n`, so `fib2`, which is `Fib(i)` at the end of each iteration, will also be `Fib(n)`.

Question 2

```
Pas(n, m)
   if m < 0 or m > n then return 0                          (1)
   else if n=0 return 1                                     (2)
   else return (Pas(n-1, m-1) + Pas(n-1, m))                (3)
```

Explanation:

Line 1 (out of the triangle): Because the leftmost column is the 0th column, any request for a number in a column below 0 must be 0. Also, because row number *n* only has up to the *n*th entry, a request for something higher than *n* (for instance, asking for row 4, column 5) must return a 0.

Line 2 (base case): If we're in the first row, and we have made it by Line 1, we must be in row 0, column 0, which is 1.

Line 3 (normal): If we are anywhere on the triangle other than row 0, column 0, we return the sum of the numbers (a) immediately above, and (b) immediately above and to the left, of the requested position

Question 3

Given positive integers *a* (`base`) and *b* (`exp`), the following algorithm finds a^b in time proportional to $\log_2 b$:

```
t = 1
n = base
m = exp

while m ≠ 1
    if m is even then
        n = n * n        (1)
        m = m/2
    else
        m = m - 1        (2)
        t = t * n
return t*n
```

Proof of correctness: Let $x = tn^m$. Immediately after the three variables *t*, *n*, and *m* are declared in the beginning of the algorithm, $t = 1$, $n = \text{base}$, and $m = \exp$. Thus

$$x = tn^m = \text{base}^{\exp}$$

In the course of the algorithm, there are two ways that the algorithm will affect *t*, *n*, or *m*:

(1) If *m* is even, then the following two operations are performed:

$$n \to n \cdot n = n^2$$
$$m \to \tfrac{m}{2}$$

We now look at how this affects *x*:

$$\begin{aligned} x &= t\left(n^2\right)^{\frac{m}{2}} \\ &= tn^{2\frac{m}{2}} \\ &= tn^m \end{aligned}$$

We see that *x* has not changed.

(2) If *x* is odd, then:

$$m \to m-1$$
$$t \to tn$$

$$\begin{aligned} x &= (tn)n^{m-1} \\ &= tn^{1+m-1} \\ &= tn^m \end{aligned}$$

In each iteration of the `while` loop, one of these two sets of operations is performed. Neither *t*, *n*, nor *m* is modified in any other way. Thus, loop invariant $x = \text{base}^{\exp}$ for the entire execution of the algorithm.

We next observe that, if $m = 1$, then

$$x = tn^1 = tn$$
$$\text{base}^{\text{exp}} = tn$$

We keep this in mind.

Examining the algorithm, we see that in every step *m* is either reduced by one or divided by two. We can express these two operations as $f(m) = \frac{m}{2}$ and $g(m) = m - 1$.

First, we notice that either of these operations results in *m* becoming smaller.

Next, we remember that only integers are allowed as input. Thus, $g(m)$ must result in an integer because 1 is an integer and the set of integers is closed under subtraction. Also, $f(m)$ is only performed on even numbers, which may always be expressed as $2i$, where *i* is an integer between 0 and *m* (having the same sign as *m*). Thus,

$$\begin{aligned} f(m) &= \frac{m}{2} \\ &= \frac{2i}{2} \\ &= i \end{aligned}$$

so $f(m)$ where *m* is positive must be a positive integer. So, applied as they are in this algorithm, both $f(m)$ and $g(m)$ will result in a positive integer, and both will decrease *m*. Thus, because of the well foundedness of positive integers, eventually *m* will become 1, the smallest positive integer. At this point execution will stop, and $x = tn^1 = tn$, so the program is correct.

Question 4

```
parSeq(n, L, R)
   if L = n then return 1                                          (1)
   if L = R then return parSeq(n, L+1, R)                          (2)
   else return (parSeq(n, L+1, R) + parSeq(n, L, R+1))             (3)
```

Explanation:

This algorithm works by breaking the problem down into smaller problems. It is called first in the form parSeq(*n*, 0, 0), where *n* is the number of sets of parentheses we are looking for combinations of. Each "instance" of the algorithm is given

(i) *n* (`n`),
(ii) The number of left parentheses "already used" (`L`), and
(iii) The number of right parentheses "already used" (`R`).

It then determines whether it is appropriate to use another left parenthesis or another right parenthesis, or whether both might be appropriate. Each possible added parenthesis is a new, smaller problem, and is sent through the algorithm again. The line-by-line interpretation follows.

Line 1 (base case): This is the smallest problem. The number of left parentheses equals the number of sets of parentheses desired. All the remaining choices must be right parentheses, so there is only one possibility. Return 1.

Line 2 (restrict to valid combinations): If there are as many right parentheses as left parentheses, then adding another right parenthesis is not allowed until a left parenthesis has been added first. Thus, the number of possible combinations is equal to the number of combinations after adding one left parenthesis. Return `parSeq(n, L+1, R)`.

Line 3 (normal): It's too big a problem for the algorithm to solve, but it can be split into two smaller problems: one in which a left parenthesis is added, and one in which a right parenthesis is added. Since adding a left parenthesis and adding a right parenthesis are the only two choices available, and each will result in a unique set of possible sequences, the total number of possible combinations is equal to the sum of the number of combinations after adding a left parenthesis and the number of combinations after adding a right parenthesis. Return `(parSeq(n, L+1, R) + parSeq(n, L, R+1))`.

Question 5

```
import java.awt.*;
import java.awt.event.*;

public class Widget extends Frame implements WindowListener
{
    int unit = 4; //The basic unit for the smallest triangle

    int Order = 8; //The starting order

  public void windowClosing(WindowEvent e) { System.exit(0); }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}

  public Widget()
    {
      Toolkit tk = Toolkit.getDefaultToolkit();
      Dimension d = tk.getScreenSize();
      int screenHeight = d.height;
      int screenWidth = d.width;

      addWindowListener(this);
      setSize(screenHeight, screenHeight);
      setLocation(screenWidth/4, screenHeight/4);
      addMouseListener(new MouseAdapter()
                    {
                     public void mouseClicked(MouseEvent evt)
                        { Order = (Order+1) % 10; repaint(); }
                    });
    }//method Widget

  public void paint(Graphics g)
    {
      g.translate(getInsets().left, getInsets().top);
      int height = getSize().height - getInsets().top - getInsets().bottom;
      int width = getSize().width - getInsets().left - getInsets().right;

      setTitle("Sierpinski Widget of order "+Order);
         g.setColor(Color.black);
         sier(g,Order,width/2,0);
       }//method paint

    private double power(int base, int exp){
        double t = 1; //This will be used to keep m even
        int m = exp;
        double n = (double) base; //We need a double because n may get large
        //right now, t*(n^m) = base^exp (because t=1, n=base, and m=exp)
        //t*(n^m) will not change anywhere in the algorithm.

        if (m == 0) return 1;

        while (m != 1) { //When m is one, we're done (t*n = base^exp)
            if (m % 2 == 0) {//Is n even?
                n = n*n; //=n^2
                m = m/2;
                //does not change t*(n^m) because:
                //t*(n^m) = t*(n^2)^(m/2)
```

```
            }
            else {//If n is not even, make it even.
                m = m - 1; //so m is divisible by two in the next iteration
                t = t * n;
                //does not change t*(n^m) because:
                //t*(n^m) = (t*n)*n^(m-1)
            }
        }
        return t*n; //base^exp = t*(n^m) = t*n (because m = 1)
    }//method power

    private void triangle(Graphics g, int x, int y){
        //Define variables to be used for other two corners of the triangle
        int x2 = 0;
        int x3 = 0;
        int y2 = 0; //the y value for both other corners is the same

        //X's are easy, because the triangle is oriented vertically, so the
        //top is, in terms of x, halfway between the other two corners
        x2 = x + this.unit/2;
        x3 = x - this.unit/2;

        //Both Y's are the same, and they can be found using the Pythagorean
        //formula. However, they must be cast to an int for drawLine()
        y2 = y + (int) java.lang.Math.sqrt(power(this.unit,2) -
power(this.unit/2,2));

        //Now, draw the lines:
        g.drawLine(x, y, x2, y2);
        g.drawLine(x, y, x3, y2);
        g.drawLine(x2, y2, x3, y2);
    }// method triangle

    public void sier(Graphics g, int order, int x, int y){
        if (order==0) return; //Don't do anything if there are no triangles!
        if (order==1) {//order=1: paint the triangle at the specified point:
            this.triangle(g, x, y);
        }
        else {
            //find L, the length of triangle whose corners will form the
            //vertices of the next three triangles:
            int L = (int) power(2, order-2) * this.unit;
            //use L to determine the location of the corners of the triangle
            //with sides of length L and vertex (x,y), using geometry and the
            //Pythagorean formula
            int xDiff = L/2;
            int yDiff = (int) java.lang.Math.sqrt(power(L,2) - power(L/2,2));
            //Draw widgets for each of the three points, with order one
            //less than the current order:
            this.sier(g, order-1, x, y);
            this.sier(g, order-1, x - L/2, y + yDiff);
            this.sier(g, order-1, x + L/2, y + yDiff);
        }
    } //method sier

    /* Please do not touch the main method. */
     public static void main (String[] args)
    {
      Widget w = new Widget();
      w.show();
    } //method main
}//class Widget
```