

# Lecture 8

## Recursion II

- Lists of Strings
  - Binary Strings
  - Combinations
- Catalan numbers
  - Formulae
  - Inductive Proof
- Fractals
- Lab 2 – Preview

# Lists of Strings

## Lists of Strings

We'll continue our investigation of recursive functions by printing out lists of strings.

- All binary strings of length  $n$ . There are  $2^n$  such strings.
- All  $(s, t)$ -combinations, which are binary strings with  $s$  copies of 0 and  $t$  copies of 1.

We'll print the strings in numeric order, which is equivalent to alphabetical order (as in a dictionary). It is also known as *lexicographic order* and it has a simple recursive definition (0s before 1s).

```
000
001
010
011
100
101
110
111
```

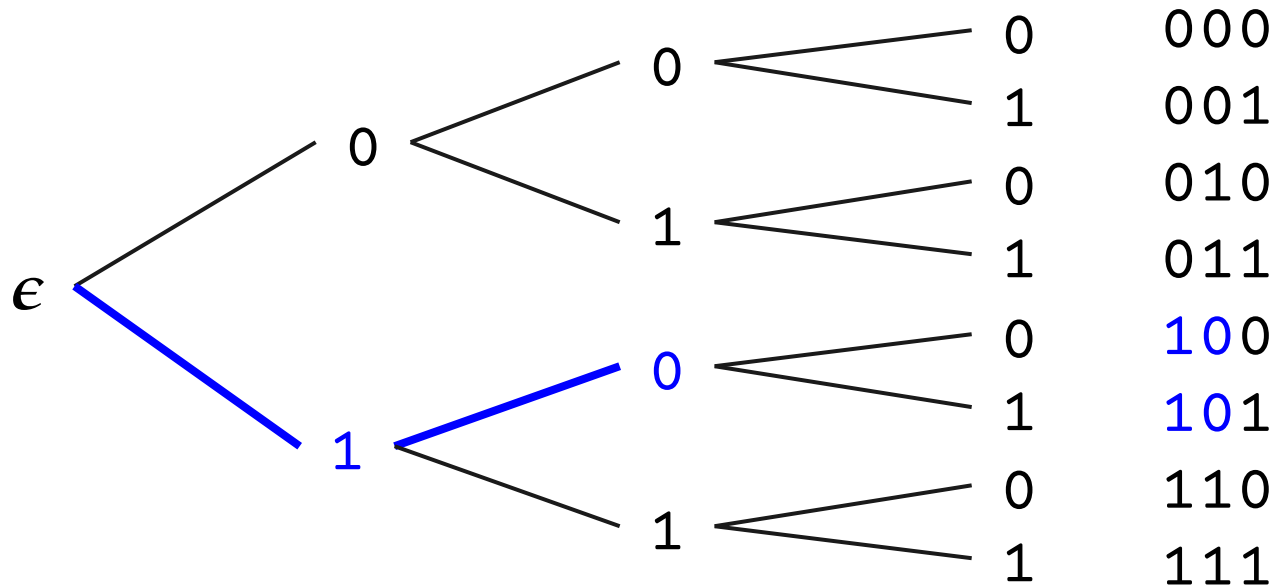
Binary strings of length  $n = 3$ .

```
00111
01011
01101
01110
10011
10101
10110
11001
11010
11100
```

$(s, t)$ -combinations for  $s = 2$  and  $t = 3$ .

How many  
 $(s, t)$ -combinations  
are there?

A program for listing binary strings is provided online as `Binary.java`. Then we'll write a new file `Combo.java` for listing combinations.



Viewing the lexicographic order of binary strings of length  $n = 3$  as a tree.

- The root is the empty string  $\epsilon$  on the left.
- Each layer branches in two ways, with 0s coming before 1s.
- Paths from the root build up the prefix for the strings that appear in the list.  
For example, all of the strings with prefix 10 appear to the right of the highlighted path.

```
GNU nano 4.8 Binary.java
public class Binary {

    public static int printAllBinary(int numBits) {
        return printAllBinaryRec("", numBits);
    }

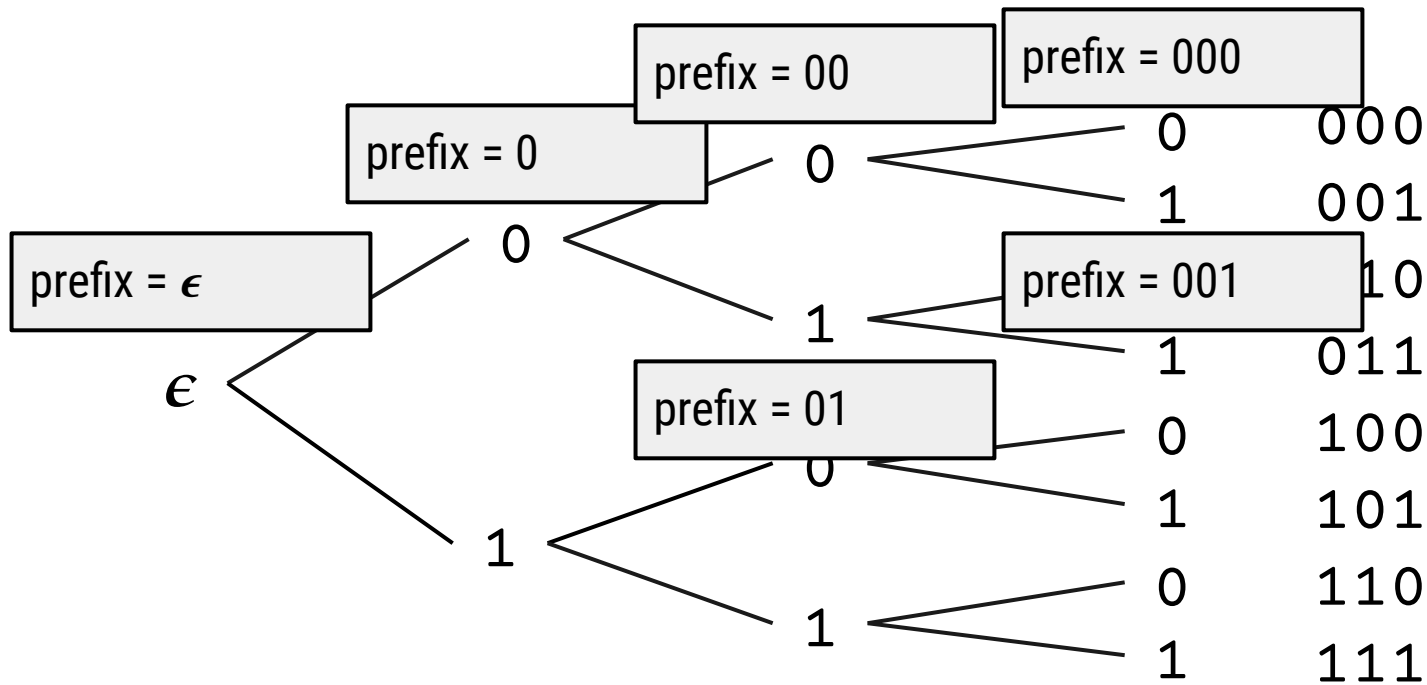
    protected static int printAllBinaryRec(String prefix, int remainingBits) {
        int total0, total1;
        if (remainingBits <= 0) {
            System.out.println(prefix);
            return 1;
        } else {
            total0 = printAllBinaryRec(prefix + "0", remainingBits - 1);
            total1 = printAllBinaryRec(prefix + "1", remainingBits - 1);
            return total0 + total1;
        }
    }

    public static void main(String[] args) {
        int n, total;
        n = Integer.parseInt(args[0]);
        total = printAllBinary(n);
        System.out.println("Total: " + total);
    }
}
```

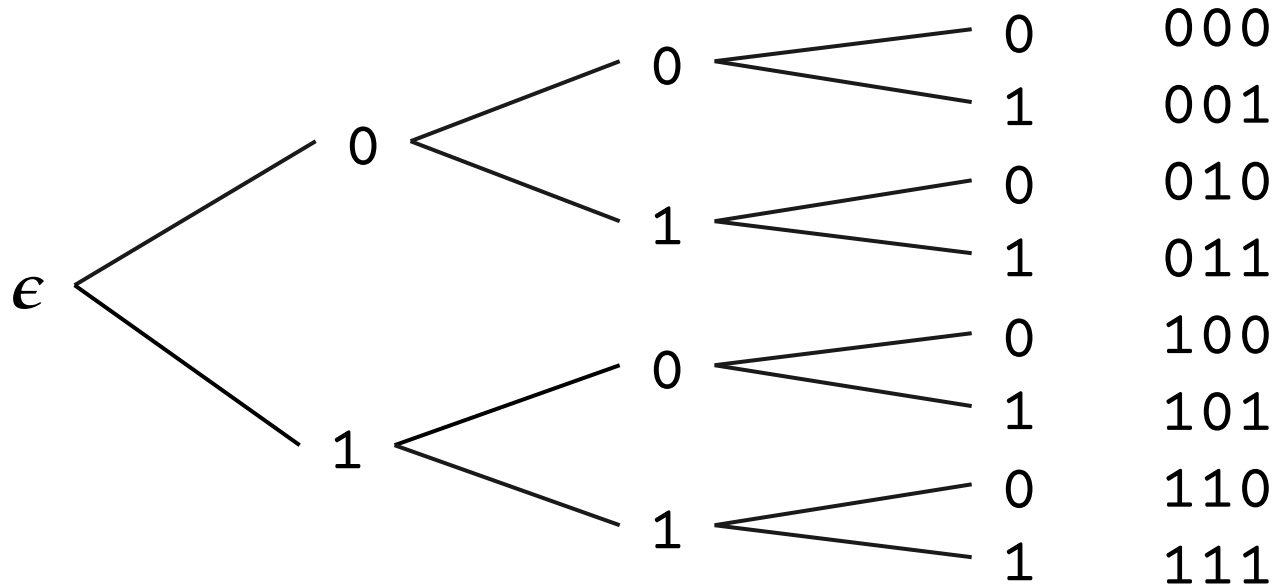
```
-> javac Binary.java
-> java Binary 3
000
001
010
011
100
101
110
111
Total: 8
->
```

Printing out all binary strings of length  $n$  in lexicographic order using `Binary.java`. In other words, all the strings starting with 0 come before those starting with 1, recursively.

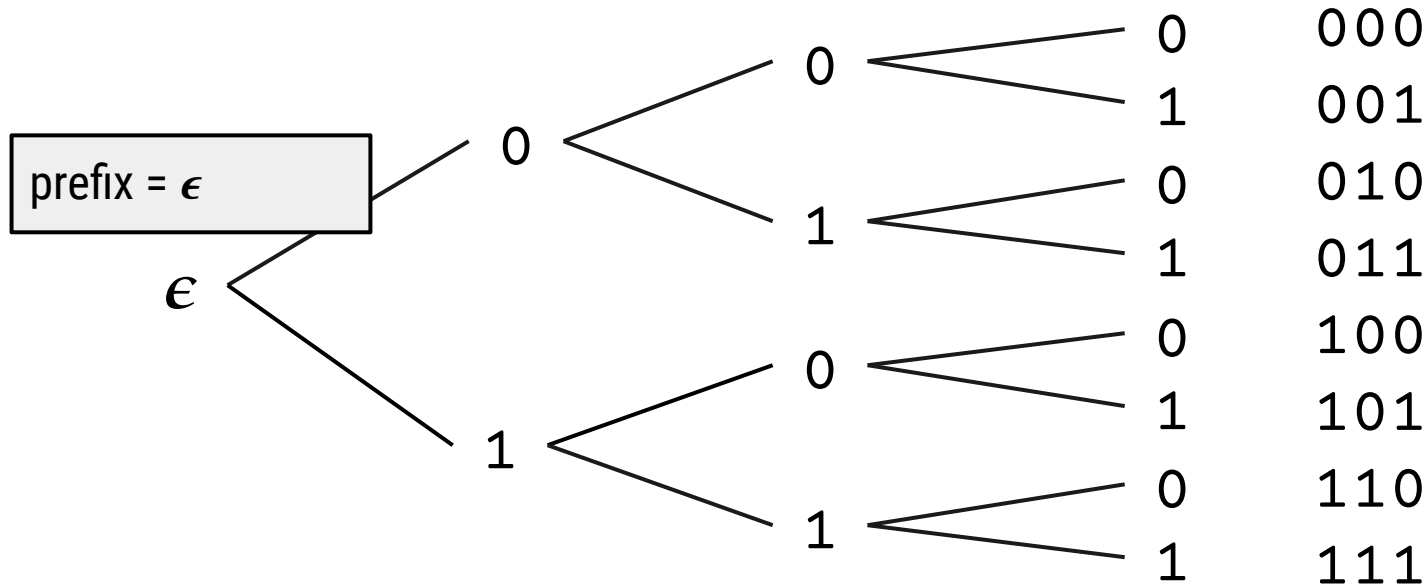
- We'll consider iterative approaches later in the course.
- What is the total for a given value of  $n$ ? How can we test our code internally?



When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)

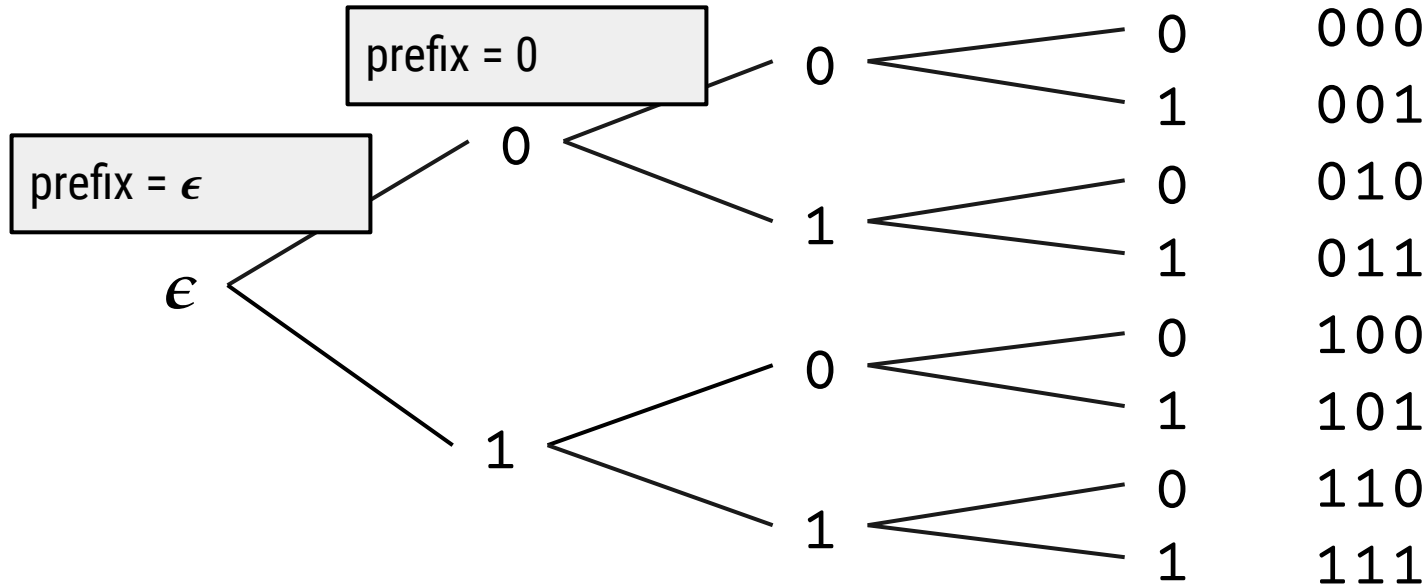


When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)

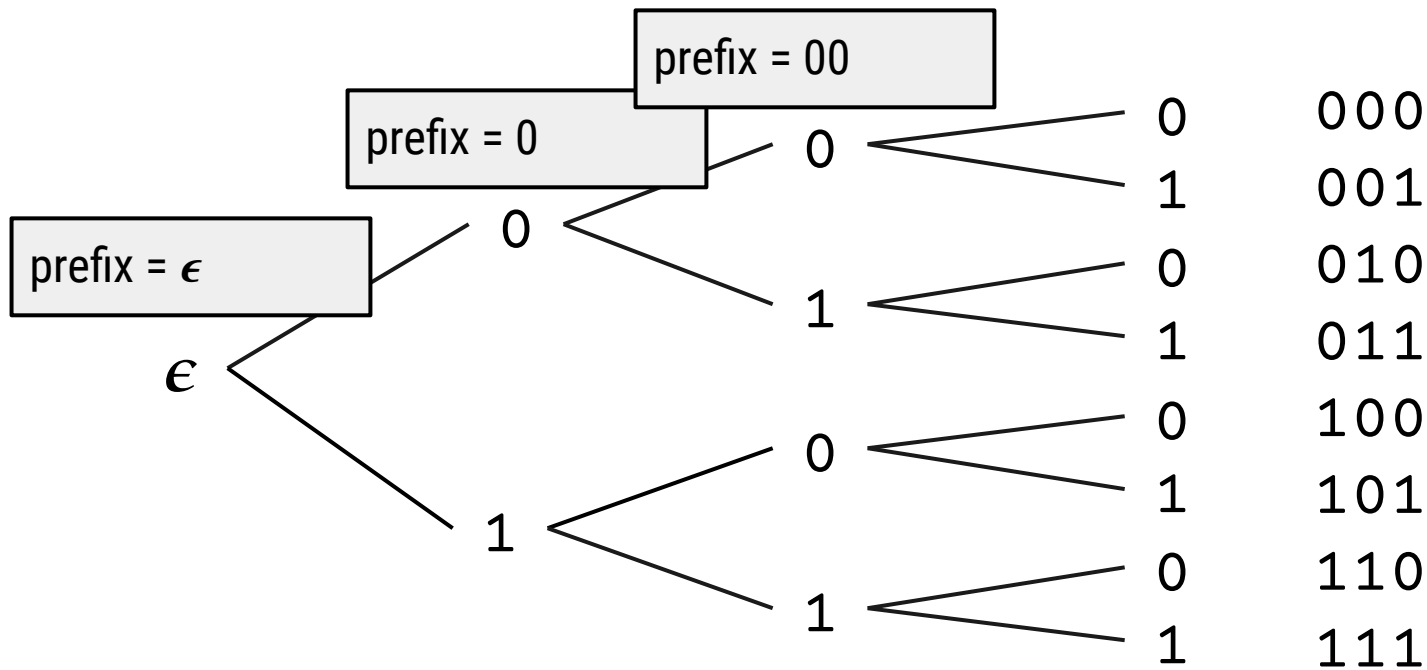


When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)

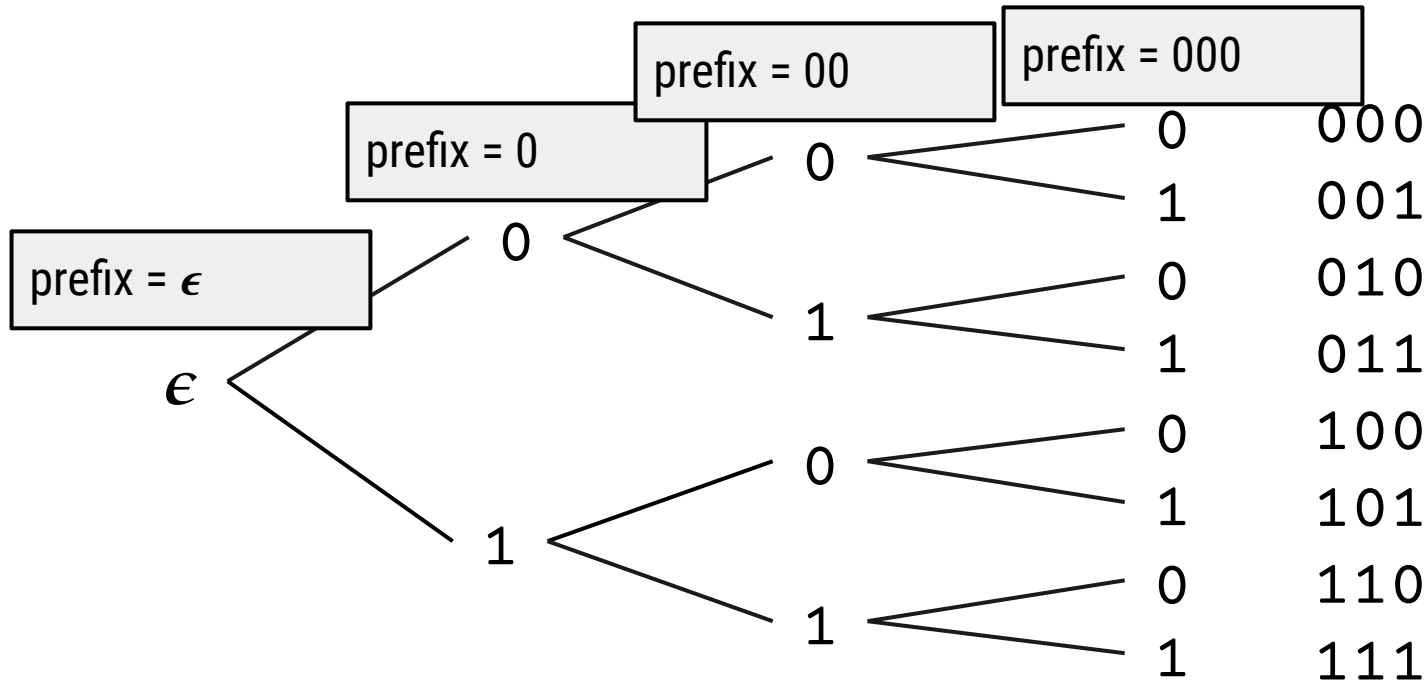




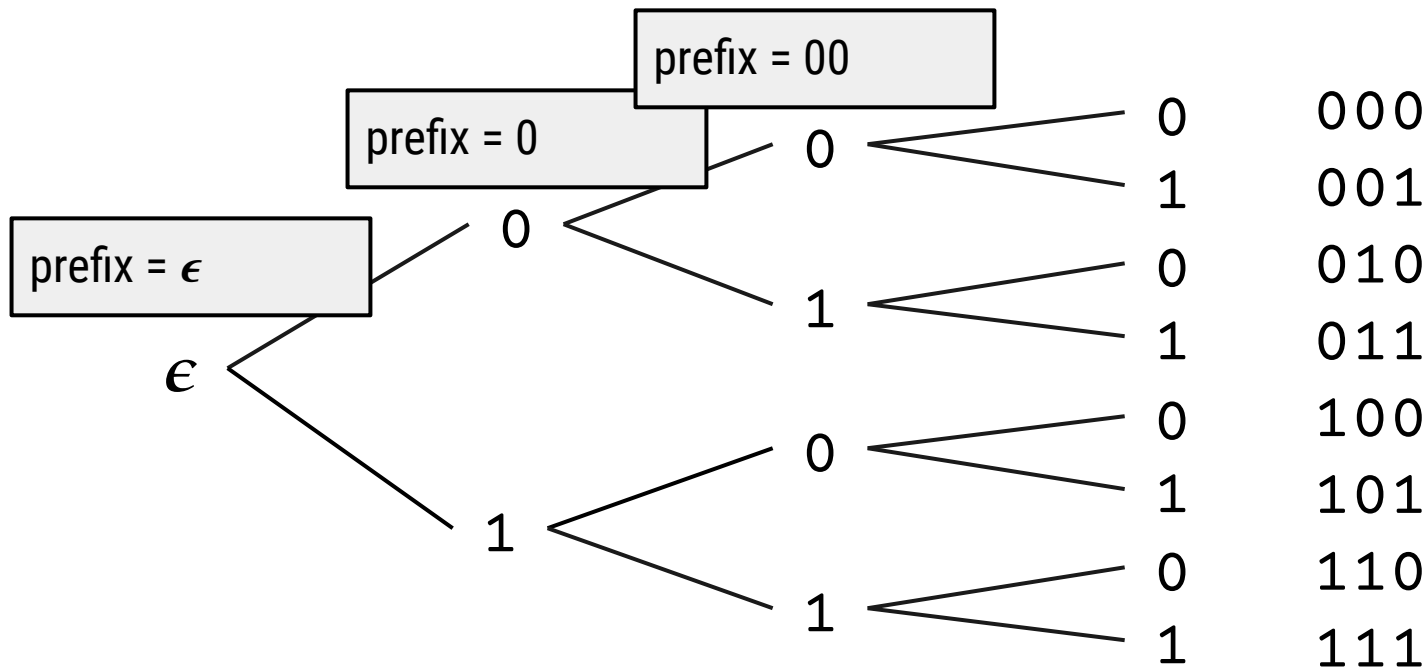
When `printAllBinary` calls itself, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)



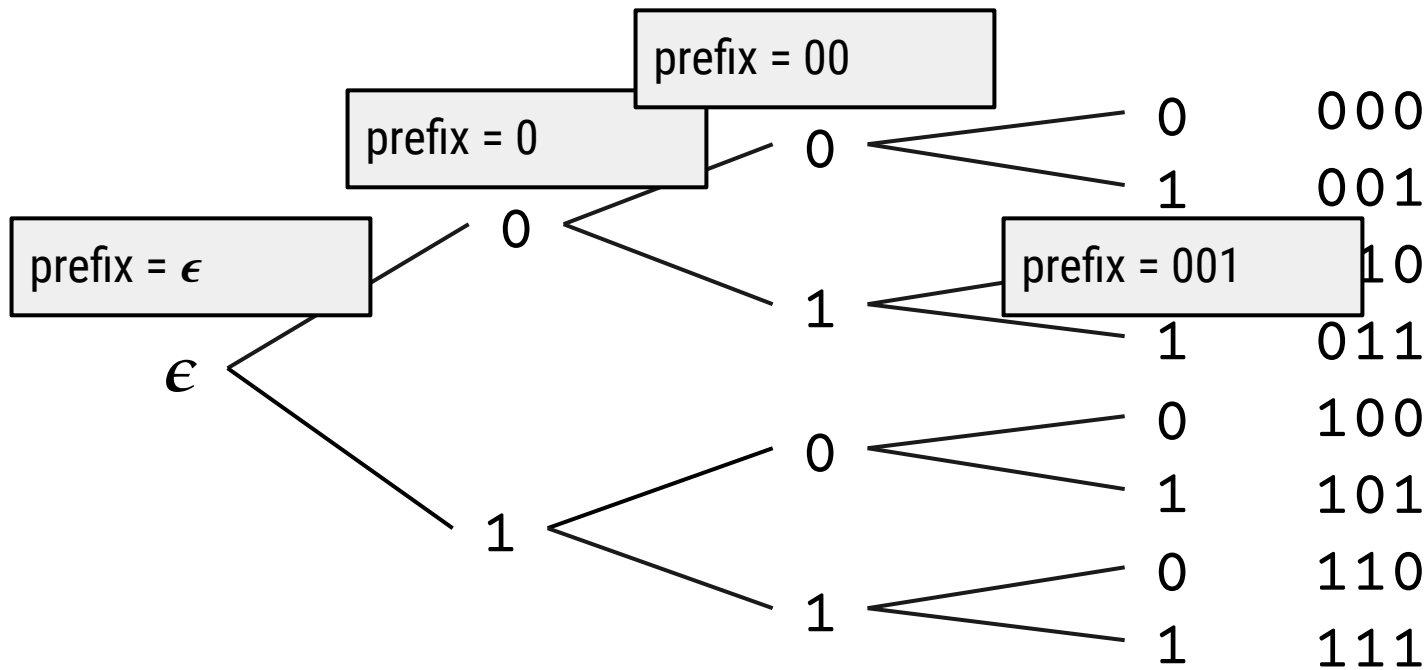
When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)



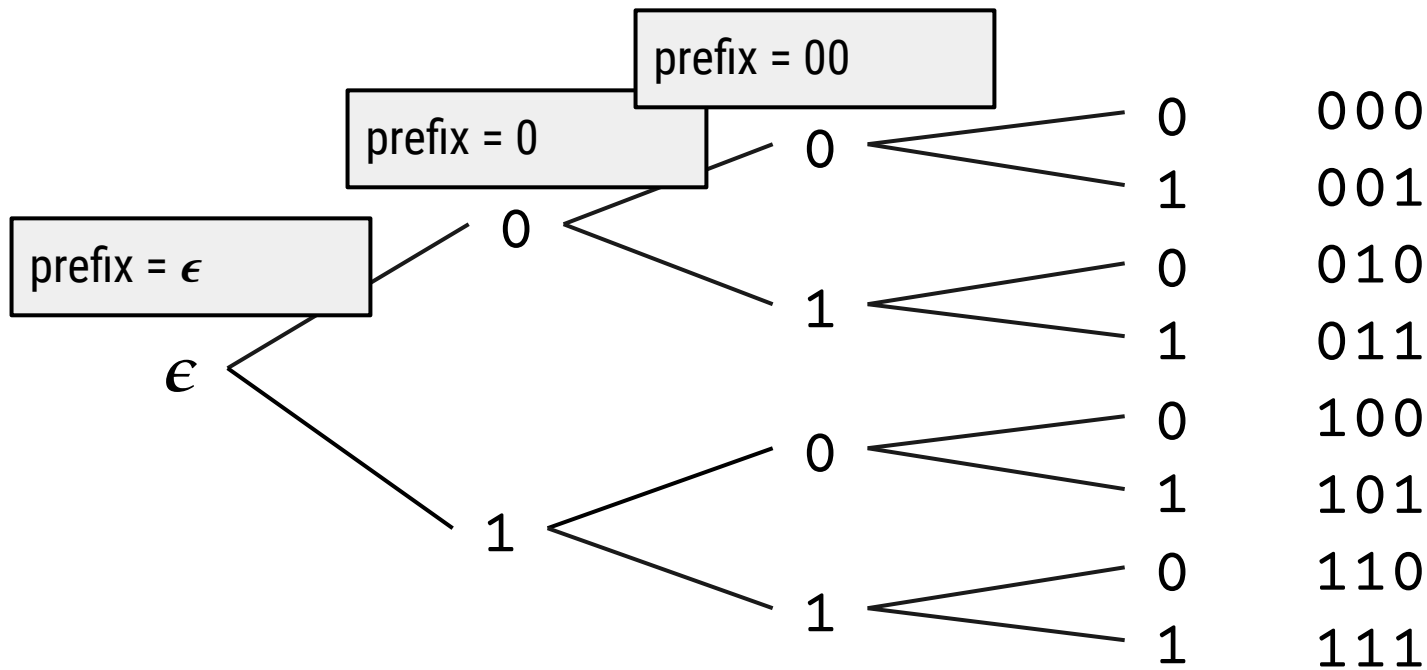
When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)



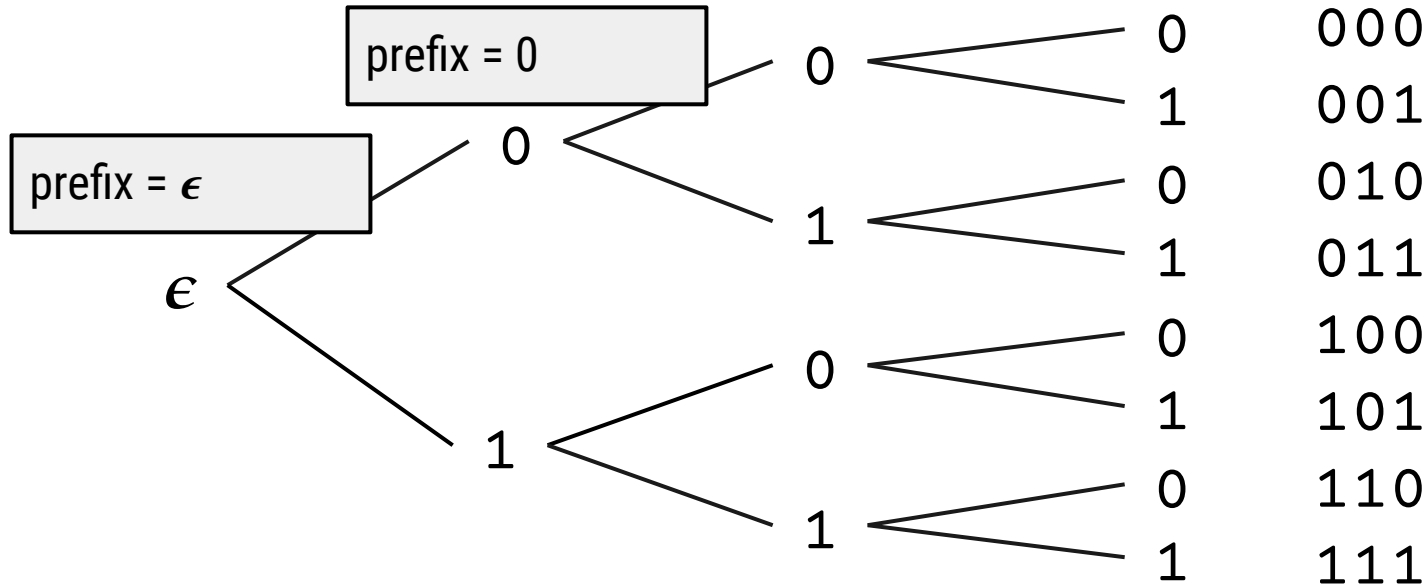
When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)



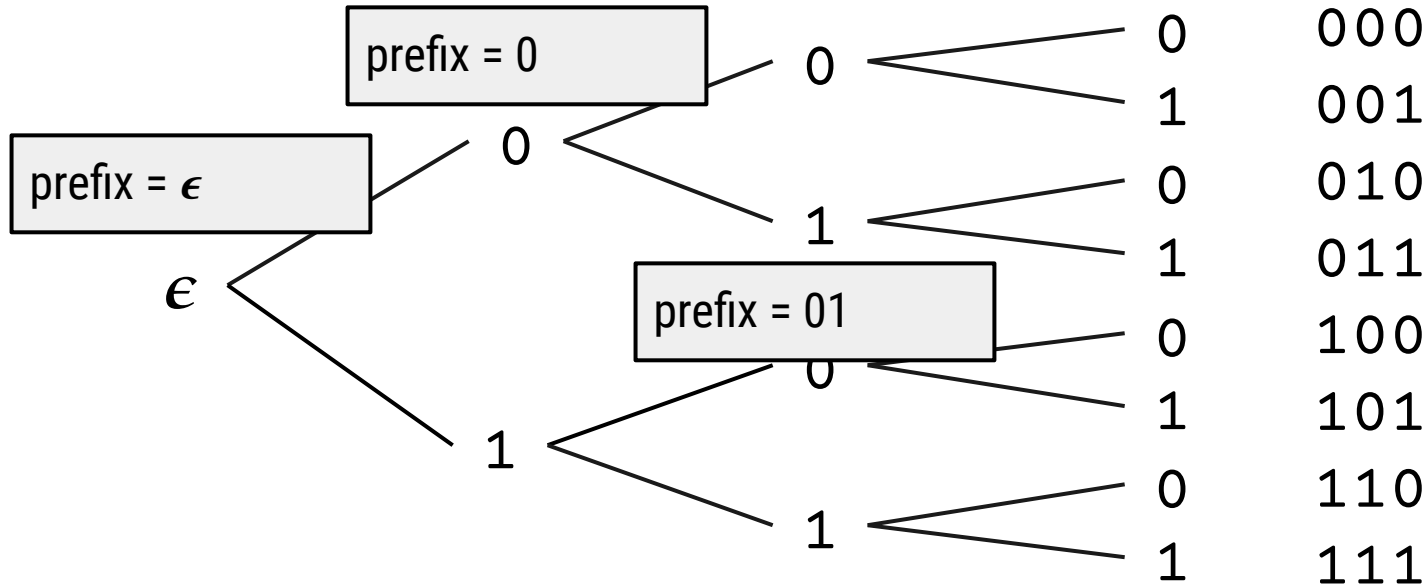
When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)



When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)

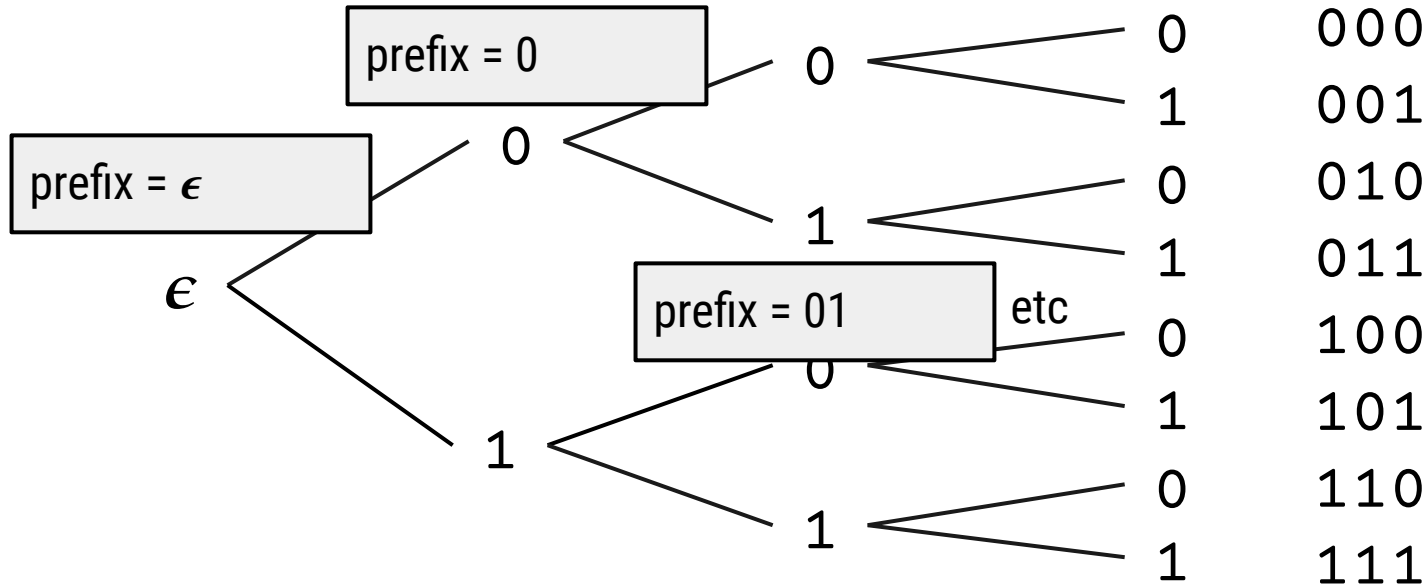


When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)



When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)





When `printAllBinary` calls its, the new instance of it is put on the top of the call stack. The evolution of the call stack is shown above, along with the value of the `prefix` parameter. (Add animations with class participation.)

```
GNU nano 4.8 Binary.java
import java.lang.Math;
import structure5.*;

public class Binary {

    public static int printAllBinary(int numBits) {
        return printAllBinaryRec("", numBits);
    }

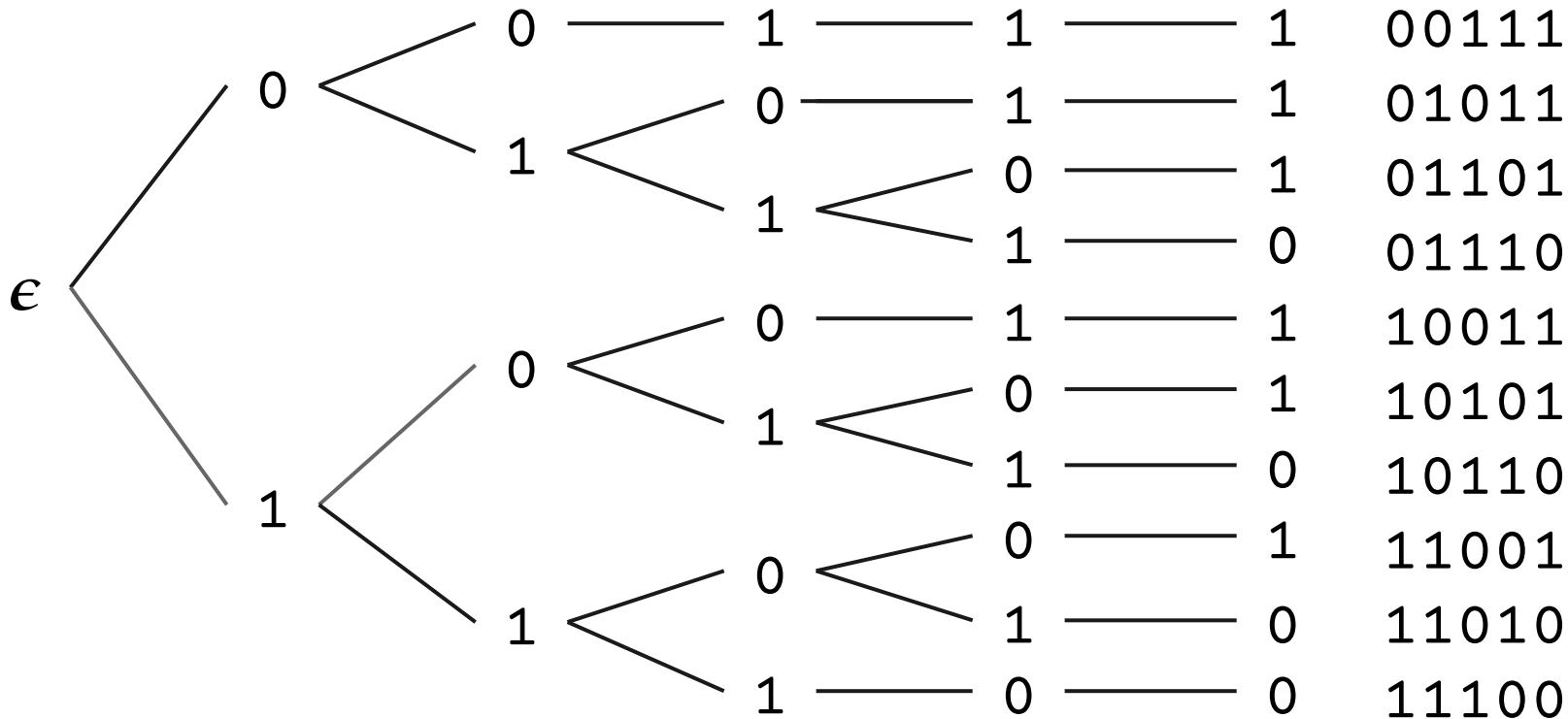
    protected static int printAllBinaryRec(String prefix, int remainingBits) {
        int total0, total1;
        if (remainingBits <= 0) {
            System.out.println(prefix);
            return 1;
        } else {
            total0 = printAllBinaryRec(prefix + "0", remainingBits - 1);
            total1 = printAllBinaryRec(prefix + "1", remainingBits - 1);
            return total0 + total1;
        }
    }

    public static void main(String[] args) {
        int n, numTotal, numCorrect;
        n = Integer.parseInt(args[0]);

        numTotal = printAllBinary(n);
        System.out.println("Total: " + numTotal);

        numCorrect = (int)Math.pow(2, n);
        Assert.condition(numTotal == numCorrect, "Expected: " + numCorrect);
    }
}
```

Updating `Binary.java` with an *assertion* to test the number generated.



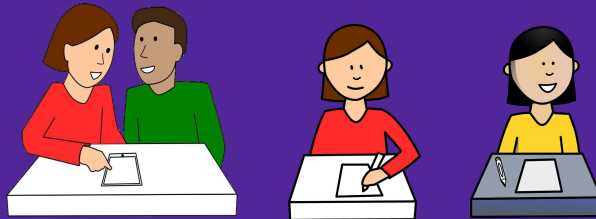
Viewing the lexicographic order of (s,t)-combinations with s = 2 and t = 3 as a tree. Notice that the nodes in this tree don't always have two branches. Why?

## Activity: Completing Combo . java

Using Binary . java as a basis, complete the file Combo . java.

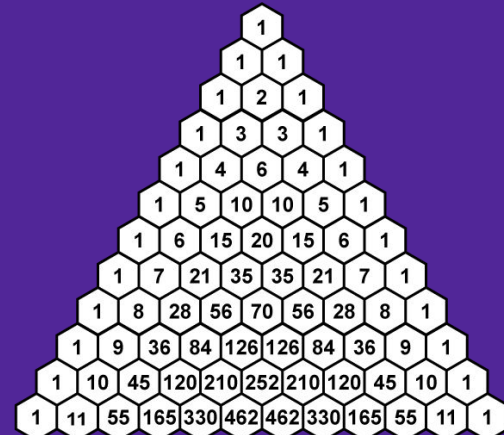
Start by talking with your neighbor. Then we'll discuss this as a group.

Finally, you'll have time to write your own version.



Discuss your ideas with a neighbor for 3 minutes.  
Then you'll have time to try writing Combo . java.

- What are your base cases?
- How many (s,t)-combinations are there?



```

GNU nano 4.8                               Combo.java

public class Combo {

    // Print out all of the (s,t)-combinations.
    public static int printAllCombo(int s, int t) {
        return printAllComboRec("", s, t);
    }

    // The helper function that does the recursive work for printAllCombo.
    // "prefix" stores the part of the string that has already been created.
    // num0 and num1 give the number of 0's and 1's that still need to be added.
    protected static int printAllComboRec(String prefix, int num0, int num1) {
        int total0, total1;
        total0 = 0;
        total1 = 0;

        // Base Case: There are no more 0s or 1s.
        if (num0 == 0 && num1 == 0) {
            System.out.println(prefix);
            return 1;
        }

        // If there are more 0s, then recursively add a 0.
        if (num0 > 0) {
            total0 = printAllComboRec(prefix + "0", num0-1, num1);
        }

        // If there are more 1s, then recursively add a 1.
        if (num1 > 0) {
            total1 = printAllComboRec(prefix + "1", num0, num1-1);
        }

        // Return the total of the recursive cases.
        return total0 + total1;
    }
}

```

```

// Computes the binomial value n choose k.
public static int binom(int n, int k) {
    // Base cases.
    if (k == 0 || n == k) {
        return 1;
    } else {
        return binom(n-1, k) + binom(n-1, k-1);
    }
}

public static void main(String[] args) {
    int s, t, numTotal, numCorrect;
    s = Integer.parseInt(args[0]);
    t = Integer.parseInt(args[1]);

    numTotal = printAllCombo(s,t);
    System.out.println("Total: " + numTotal);

    numCorrect = binom(s+t, t);
    Assert.condition(numTotal == numCorrect, "Expected: " + numCorrect);
}

```

```

-> javac Combo.java
-> java Combo 2 3
00111
01011
01101
01110
10011
10101
10110
11001
11010
11100
Total: 10

```

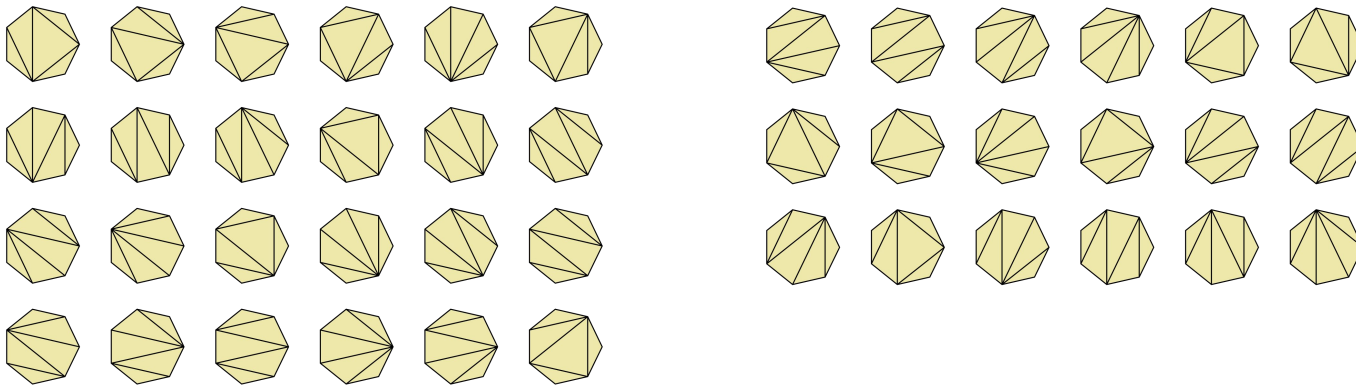
# Catalan Numbers

# Number of Triangulations of Convex Polygons

Let  $C(n)$  be the  $n^{\text{th}}$  [Catalan number](#). The Catalan sequence starting with  $n = 0$  is [OEIS 108](#):

1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

**Theorem:** The number of different triangulations of a convex polygon with  $n$  vertices is  $C(n-3)$ .



The triangulations of a heptagon with  $n = 7$  vertices.

There are a total of  $C(4) = 42$  such triangulations.

The Catalan numbers come up frequently in computer science.

- $C(n)$  is the number of binary trees with  $n$  internal nodes.
- $C(n)$  is the number of balanced parentheses strings of length  $2n$ .

For example, when  $n = 3$ , the five such strings are  $((()))$ ,  $(()())$ ,  $(())()$ ,  $()(())$ ,  $()()()$ .

## Balanced Parentheses

A string of parentheses (e.g. “(“ and “)”) is *balanced* or *well-formed* or *properly nested* if the following two properties hold.

1. There are  $n$  copies of “(“ and  $n$  copies of “)” for some  $n$ .
2. No prefix contains more “(“ than “)”.

For example, the balanced parentheses with  $n = 3$  pairs are as follows:

((())) ()(()) ()() (())() (())()

These strings are often represented in binary.

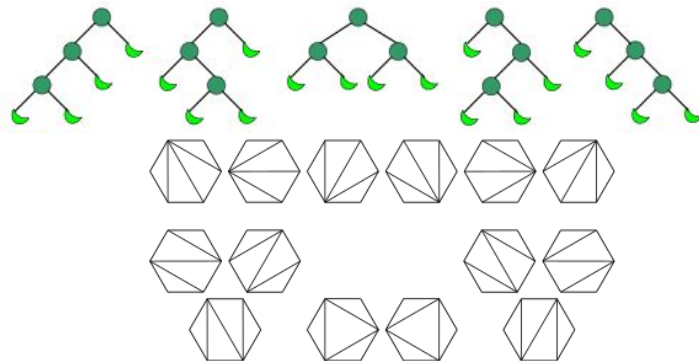
111000 101100 101010 110010 110100

Balanced parentheses of length  $2n$  are counted by the Catalan numbers.

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^n \frac{n+k}{k} \quad \text{for } n \geq 0.$$

These numbers also count hundreds of other important mathematical objects.

- Binary trees with  $n$  internal nodes.
- Diagonalizations of  $n+3$ -gons.





# Formulae

# Catalan Formulae

There are several different formulae for Catalan numbers.

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \prod_{k=2}^n \frac{n+k}{k} \quad \text{for } n \geq 0.$$

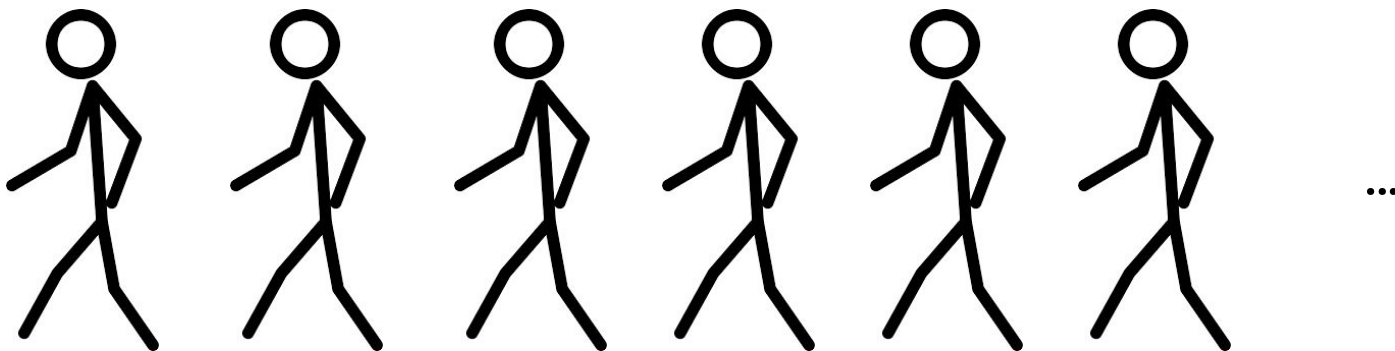
$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{for } n \geq 0$$

The top formulae are closed form

Which formula would you use?

- Are there any numeric issues?
- What about efficiency?
- Which formula is recursive?

# Inductive Proofs



Consider the following situation.

- The first person in the line (i.e., the person on the left) is named Oscar.
- If the  $i$ th person in line is named Oscar, then the  $(i+1)$ st person is named Oscar.

What can we conclude? Why?

Consider variations of the above points, and what we can conclude.

## Sum of Odd Numbers

Consider the sum of the first  $n$  odd numbers starting from 1.

- $n = 1$ . The sum is 1.
- $n = 2$ . The sum is  $1 + 3 = 4$ .
- $n = 3$ . The sum is  $1 + 3 + 5 = 9$ .
- $n = 4$ . The sum is  $1 + 3 + 5 + 7 = 16$ .

Do you notice a pattern?

It looks like the sum is  $n^2$ .

## Proof by Induction

Let  $S(n)$  be the sum of the first  $n$  odd numbers starting from 1. That is,  $S(n) = 1 + 3 + \dots + 2n-1$ .

**Theorem:**  $S(n) = n^2$  for all  $n \geq 1$ .

**Proof:** We will prove that the statement is true by mathematical induction on  $n$ .

Base Case:  $n = 1$ . In this case,  $S(1) = 1$  is the sum of the first 1 odd number starting from 1. The statement of the theorem claims  $S(1)$  equals  $1^2 = 1$ . This is true, so the base case is true.

Inductive Assumption: Assume that the statement is true for some  $k$  where  $k \geq 1$ .

Inductive Conclusion: Now we must prove that the statement is true for  $n = k + 1$ .

The sum of the first  $k+1$  odd numbers starting from 1 is


$$S(k+1) = 1 + 3 + \dots + 2(k+1)-1 = S(k) + 2(k+1)-1.$$

By the inductive assumption,  $S(k) = k^2$ . Therefore, we can substitute in this value and continue.

$$= k^2 + 2(k+1)-1 = k^2 + 2k + 2 - 1 = k^2 + 2k + 1 = (k+1)^2.$$

Therefore,  $S(k+1) = (k+1)^2$ . This proves that the statement is true for  $n = k + 1$ .

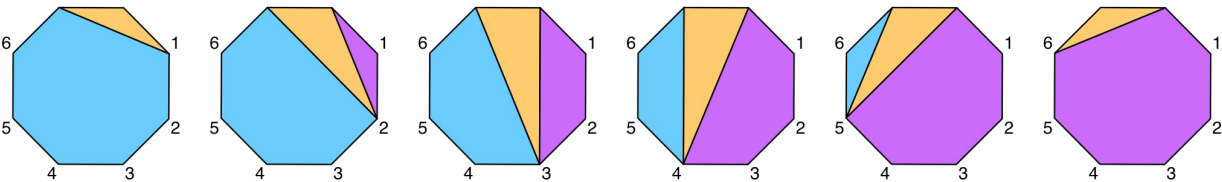
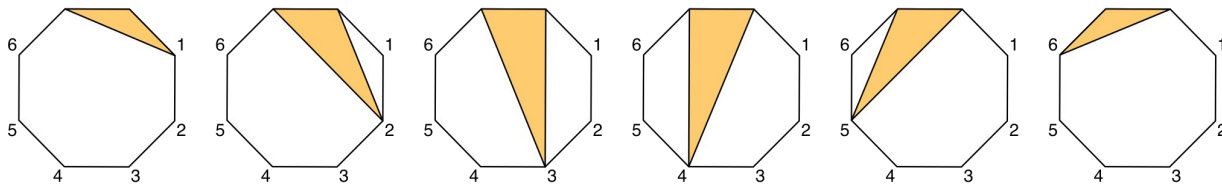
Therefore, by the principles of mathematical induction, the theorem is true for all  $n \geq 1$ .



This is about the level of difficulty for an inductive proof in this course.

# Counting Triangulations

**Theorem:** The number of triangulations of a convex polygon with  $n$  vertices is  $C_{n-3}$  for all  $n \geq 3$ .



$$C_0 = 1$$

$$C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{for } n \geq 0$$

The idea of the proof is illustrated above.

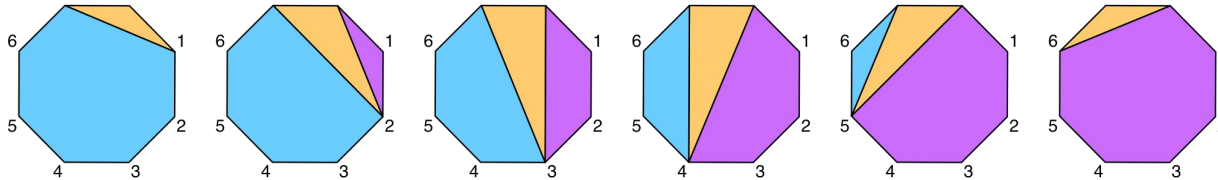
The top line must be part of a triangle (in orange).

Once this triangle is chosen, the problem is broken into two subproblems that are solved independently.

Note: The following inductive proof is more difficult than a standard CSCI 136 proof, and is more in line with CSCI 256.

# Inductive Proof

**Theorem:** The number of triangulations of a convex polygon with  $n$  vertices is  $C_{n-2}$  for all  $n \geq 2$ .



$$C_0 = 1$$

$$C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{for } n \geq 0$$

**Proof:** We will prove the theorem's statement is true for all  $n \geq 2$  by induction on  $n$ .

- **Base Case:  $n=2$ .** The convex polygon is a line, which we consider to have 1 triangulation. The statement says that there are  $C_{2-2} = C_{0} = 1$  triangulations. Therefore, the statement is true for this base case.
- **Base Case:  $n=3$ .** The convex polygon is a triangle, so it has 1 triangulation. The statement says that there are  $C_{3-2} = C_{1} = 1$  triangulations. Therefore, the statement is true for this base case.
- **Inductive Assumption:** Assume that the statement is true for all  $n$  satisfying  $2 \leq n \leq k$  for a specific  $k \geq 3$ . In other words, we assume that the theorem's statement is true for  $n = 3, 4, \dots, k$  for some value of  $k \geq 3$ .
- **Inductive Conclusion:** Consider the statement for  $n = k + 1$  where  $k \geq 3$ . The convex polygon has  $k+1$  sides and  $k+1 \geq 4$ . In each triangulation of this polygon, the top line is included in a triangle in  $k-1$  different ways. The other two lines of this triangle divide the polygon into two convex polygons. If the left polygon has  $i$  sides where  $i \geq 2$ , then the right polygon has  $k - i$  sides. (Shown above: The  $k = 7$  case. A **triangle** splits the  $k+1$ -sided polygon into **left** and **right** in all possible  $k-1 = 6$  ways.) By induction, the left polygon can be triangulated in  $C_{i-2}$  ways, and the right can be triangulated in  $C_{k+2-i}$  ways. The two triangulations are independent, so this gives a total of  $C_{i-2} \cdot C_{k+2-i}$  triangulations when the left has  $i$  sides. Summing over all  $i = 2, 3, \dots, k-1$  gives the recursive formula with the summation shown above. Therefore, the statement is true for  $n = k+1$ .

Therefore, by the principles of mathematical induction, the statement is true for all  $n \geq 2$ .



## Activity: The Number of Binary Trees

How can you prove that the binary trees with  $n$  nodes are enumerated by the Catalan numbers?



Discuss your ideas with a neighbor for 3 minutes.  
Then we'll discuss it as a group.


- What is the statement of the theorem?
- What are your base cases?
- What is your inductive assumption?
- How do you handle the inductive conclusion?

Hint: Try to do something similar to the triangulations proof.

# Fractals

# Fractals

Fractals often have recursive definitions.

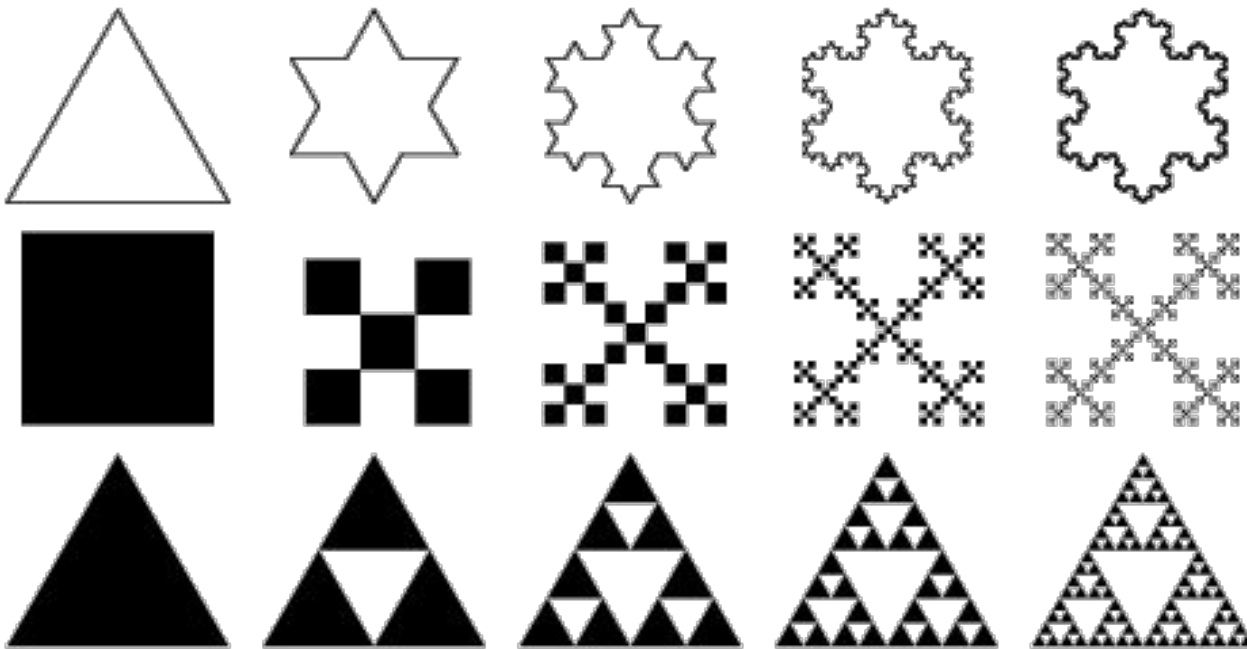
 **frac·tal**  
*/ˈfraktəl/*

**MATHEMATICS**

*noun*  
a curve or geometric figure, each part of which has the same statistical character as the whole. Fractals are useful in modeling structures (such as eroded coastlines or snowflakes) in which similar patterns recur at progressively smaller scales, and in describing partly random or chaotic phenomena such as crystal growth, fluid turbulence, and galaxy formation.

*adjective*  
relating to or of the nature of a fractal or fractals.  
"fractal geometry"

Definitions from Oxford Languages *Feedback*



Provide recursive definitions for each row.

# Lab 2 – Preview

## Computer Science CS136 (Fall 2021)

*Duane Bailey & Aaron Williams*

Laboratory 2

*Recursion*

**Objective.** Practice solving problems using recursion.

**Discussion.** This week's lab is structured as several small problems that can be solved in isolation. Recursion can be a difficult concept to master and one that is worth concentration on separately before using it in large programs. Recursive solutions can often be formulated in just a few concise, elegant lines, but they can be very subtle and hard to get right.

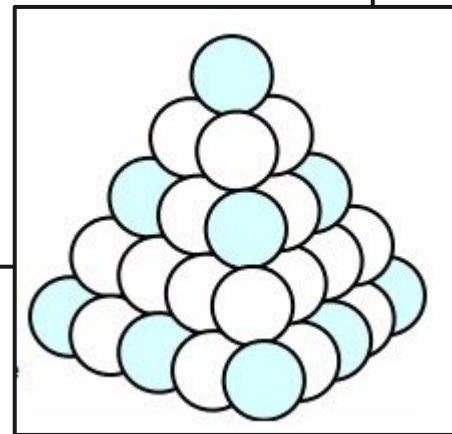
*We would like you to write this week's problem solutions using recursion.* Take time to figure out how each problem is self-referential in nature and how you could formulate the solution to the problem if you already had the solution to a smaller, simpler version of the problem. You will often need to have faith that your solution will ultimately be correct, even early in the design of the code. Remember, recursion requires three things: one or more base cases, a reduction to a subproblem, and a little progress that glues the solutions to the subproblem together. If you learn to think recursively, the solutions to many problems will seem very intuitive.

In Lab 2 you will implement a handful (or more) functions recursively.

- You will also provide test functions that help ensure that your implementation is correct. You won't print out the results, but rather check that no assertions are triggered.

4. Write a method, `tetrahedral(n)`, that computes the number of cannonballs that are stacked in a tetrahedral pyramid with  $n$  layers. The top has one cannonball, which rests on a layer with  $1 + 2 = 3$  cannonballs. This layer sits atop a layer with  $1 + 2 + 3 = 6$  cannonballs. The pattern continues. Here are some example values:

method call	result
<code>tetrahedral(0)</code>	0
<code>tetrahedral(1)</code>	1
<code>tetrahedral(2)</code>	4
<code>tetrahedral(3)</code>	10



A sample question from the lab.