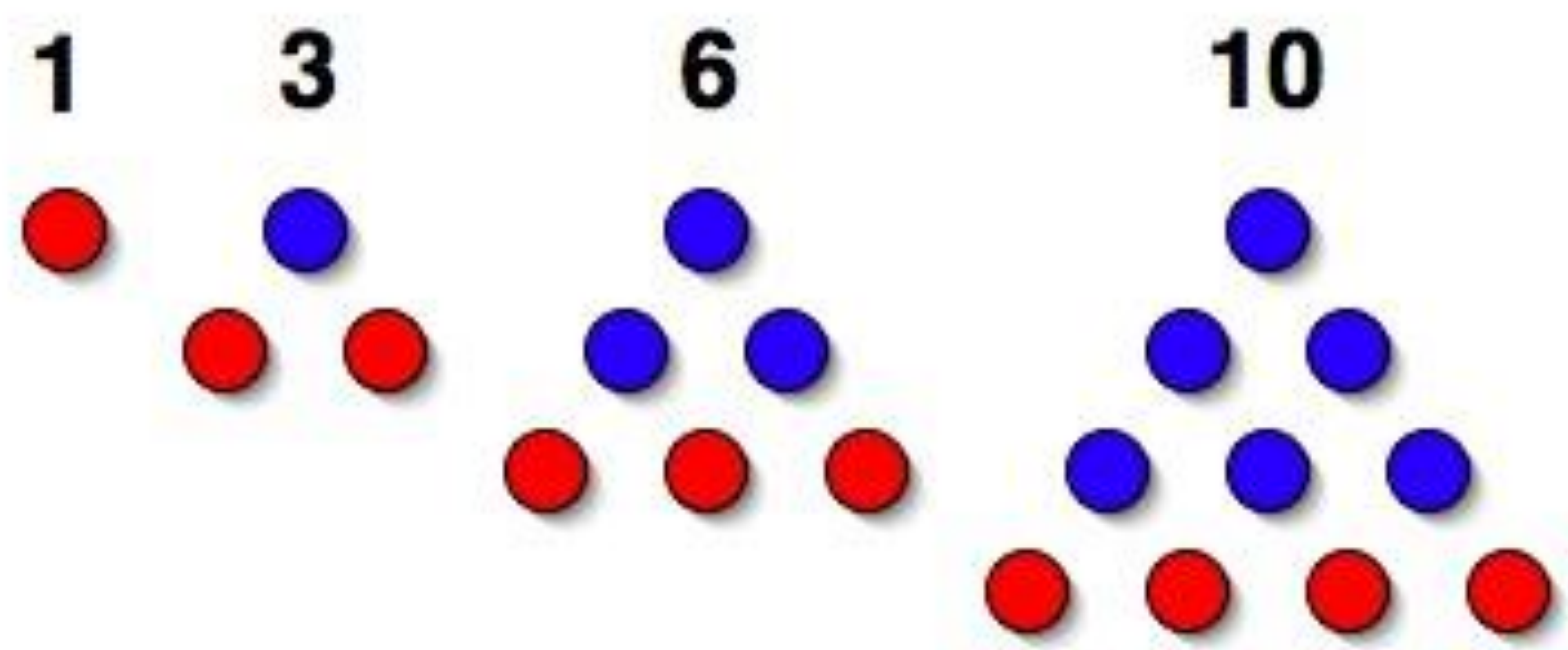
Data Structures & Algorithms

Adil M. Khan
Professor of Computer Science
Innopolis University

Recursion

Triangular Numbers

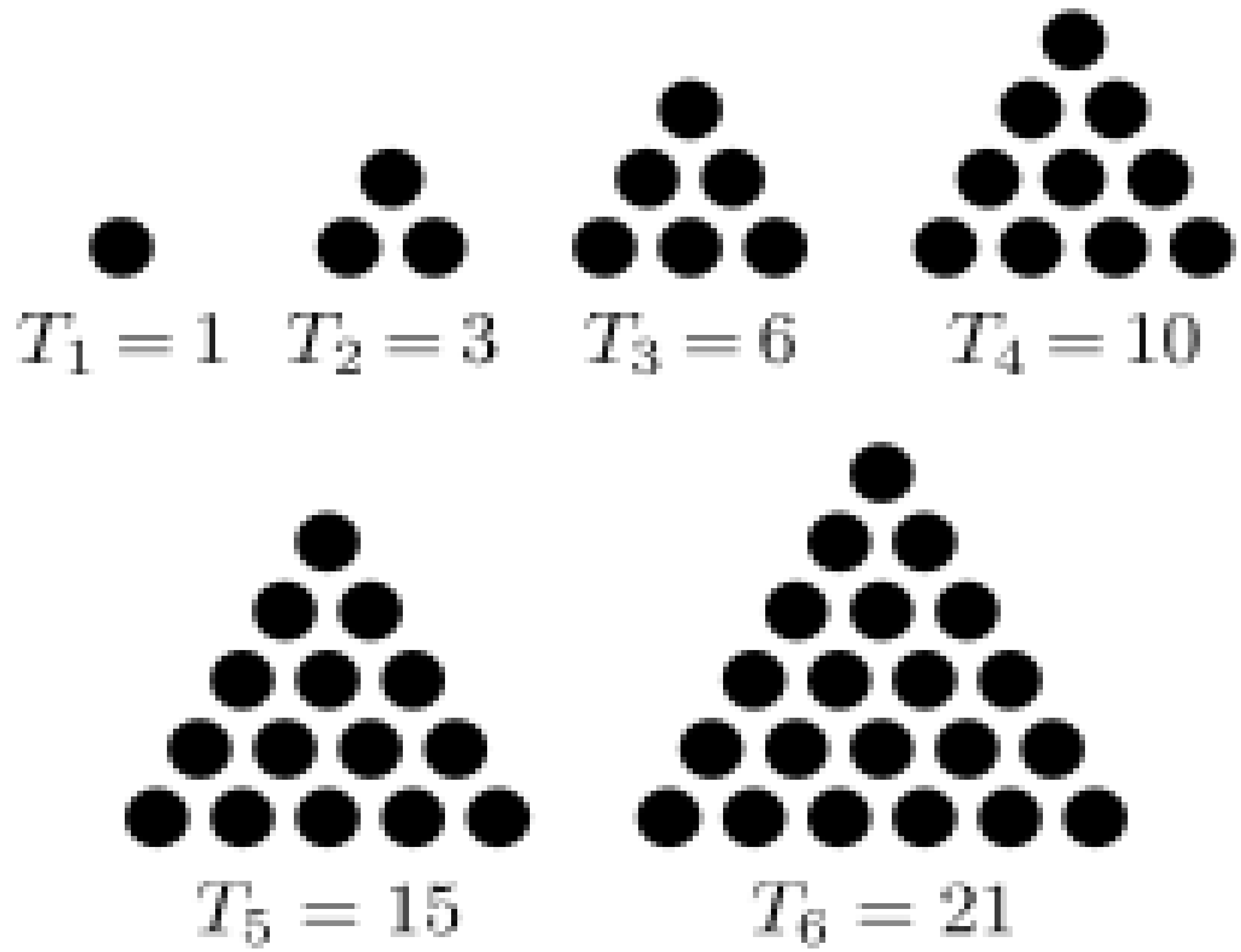


Triangular Number

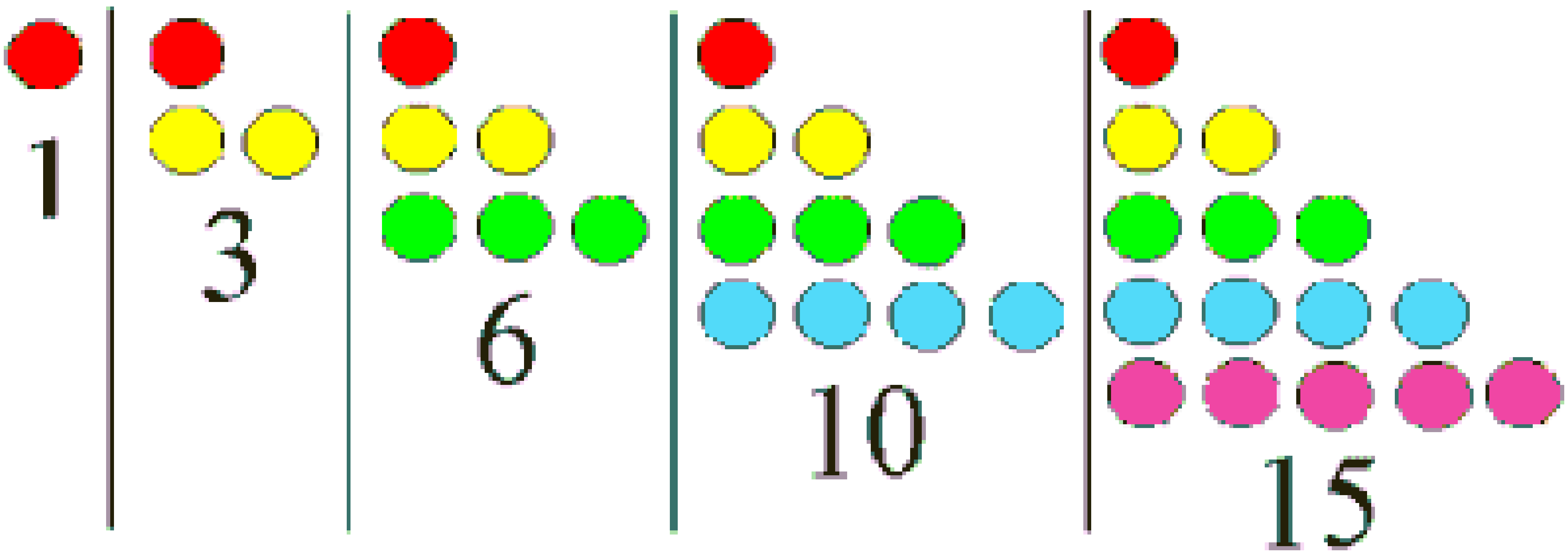
- Suppose you want to find the n^{th} triangular number
- How would you calculate it?

Triangular Numbers

The nth term in the series is obtained by adding n to the previous term.



Triangular Numbers as Columns of Circles



T(5) = 5 + sum of circles in remaining 4 columns

Triangular Numbers as Columns of Circles

T(5) = 5 + sum of circles in remaining 4 columns

Sum of circles in remaining 4 columns = ?

Triangular Numbers as Columns of Circles

T(5) = 5 + sum of circles in remaining 4 columns

Sum of circles in 4 columns = T(4) = 4 + sum of circles in remaining 3 columns

Thus,

T(5) = 5 + T(4) = 5 + 4 + T(3) = 5 + 4 + 3 + T(2) = 5 + 4 + 3 + 2 + **T(1)** = 5 + 4 + 3 + 2 + 1

Triangular Numbers

```
int triangle(int n) {

    if(n==1)
        return 1;

    else
        return( n + triangle(n-1) );
}
```

Recursive Programming

- Many problems can be ***elegantly*** described using recursion

- Writing functions recursively usually results in smaller, more concise, elegant and easier-to-understand code

- note, **elegant** does not necessarily mean **efficient**.

Understanding Recursion

- Defining something in terms of itself
- For example, let's look at the following definition of Natural Numbers
  - *0 is a natural number*
  - *if n is a natural number then so is n+1*

As an exercise, provide a similar definition for Odd Integers!

Another Example

- Group of People
  - *A group is 2 people*
  - *If "n" is a group then so is "n+1 more person"*

As an exercise, provide a similar definition for "A Sequence of Characters"!

Another Example

- Ancestors of x
  - *Mother(x) is in Ancestors(x)*
  - *Father(x) is in Ancestors(x)*
  - *If “y” is in Ancestors(x) then so are Mother(y) and Father(y)*

As an exercise, provide a similar definition for “Descendants of a Person”!

Recursive Programming

- To program recursively, you must learn to view the problem as a **BIG PROBLEM**, made from **SMALLER PROBLEMS** of exactly the same type as the big problem

- This strategy of solving problems is called "**divide-and-conquer**" strategy

Recursive Programming

- Divide
  - Break the problem into several problems that are similar to the original problem but smaller in size
- Conquer
  - Solve the smaller problems recursively, or
  - <u>If they are small enough, solve them directly</u>
- Combine the solutions to the sub-problems into a solution of the original problem

Recursive Functions

- Thus, the recursive definition of a function consists of
  - ***The base case:*** the starting point
  - ***The recursive part:*** all the other cases in terms of smaller versions of itself

Exercise

- Identify the “base cases” and the “recursive cases” in the recursive definitions (previous slides) of:
  - Natural numbers
  - Odd numbers
  - Ancestors of a person
  - Descendants of a person
  - A group of people
  - A sequence of characters

Recursive Functions

- Remember, to define a function recursively, we must
  - have a base case
  - ensure that each recursive call progresses towards the base case

Examples
(Palindromes)

```
public static boolean isPalindrome(String s) {
   return (s.length() <= 1) ||
      (s.charAt(0) == s.charAt(s.length()-1) &&
         isPalindrome(s.substring(1, s.length()-1)));
}
```

```
isPalindrome("racecar")
  = ('r' == 'r') && isPalindrome("aceca")
  = true && isPalindrome("aceca")
  = ('a' == 'a') && isPalindrome("cec")
  = true && isPalindrome("cec")
  = ('c' == 'c') && isPalindrome("e")
  = true && isPalindrome("e")
  = true
```

Examples

(Factorial)

```
public static long factorial(int n) {
    return (n <= 0) ? 1 : n * factorial(n - 1);
}
```

```
factorial(4)
  = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * (3 * (2 * factorial(1)))
  = 4 * (3 * (2 * (1 * factorial(0))))
  = 4 * (3 * (2 * (1 * 1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

Examples

(Log)

```
public static int log(int n) {
    if (n <= 0) {
        throw new IllegalArgumentException();
    } else if (n == 1) {
        return 0;
    } else {
        return 1 + log(n / 2);
    }
}
```

```
log(100)
  = 1 + log(50)
  = 1 + 1 + log(25)
  = 1 + 1 + 1 + log(12)
  = 1 + 1 + 1 + 1 + log(6)
  = 1 + 1 + 1 + 1 + 1 + log(3)
  = 1 + 1 + 1 + 1 + 1 + 1 + log(1)
  = 1 + 1 + 1 + 1 + 1 + 1 + 0
  = 6
```

Examples

(Sum of Digits)

```
public static int sumOfDigits(int n) {
    if (n < 0) {
        return sumOfDigits(-n);
    } else if (n < 10) {
        return n;
    } else {
        return sumOfDigits(n / 10) + (n % 10);
    }
}
```

sumOfDigits(-48729)
= sumOfDigits(48279)
= sumOfDigits(4827) + 9
= (sumOfDigits(482) + 7) + 9
= ((sumOfDigits(48) + 2) + 7) + 9
= (((sumOfDigits(4) + 8) + 2) + 7) + 9
= (((4 + 8) + 2) + 7) + 9
= ((12 + 2) + 7) + 9
= (14 + 7) + 9
= 21 + 9
= 30

Examples

(Fibonacci Number)

```
public static BigInteger fibonacci(int n) {
        if (n < 0) {
            throw new IllegalArgumentException();
        }
        return n <= 1 ? BigInteger.valueOf(n)
                    : fibonacci(n-1).add(fibonacci(n-2));
    }
```

```
fib(4)
  = fib(3) + fib(2)
  = (fib(2) + fib(1)) + fib(2)
  = ((fib(1) + fib(0)) + fib(1)) + fib(2)
  = ((1 + fib(0)) + fib(1)) + fib(2)
  = ((1 + 0) + fib(1)) + fib(2)
  = (1 + fib(1)) + fib(2)
  = (1 + 1) + fib(2)
  = 2 + fib(2)
  = 2 + (fib(1) + fib(0))
  = 2 + (1 + fib(0))
  = 2 + (1 + 0)
  = 2 + 1
  = 3
```

Examples

(Greatest Common Divisor)

```
public static int gcd(int a, int b) {
        if (a < 0 || b < 0) {
            throw new IllegalArgumentException("No GCD of negative integers");
        }
        return b == 0 ? a : gcd(b, a % b);
    }
```

```
gcd 444 93
  = gcd 93 72
  = gcd 72 21
  = gcd 21 9
  = gcd 9 3
  = gcd 3 0
  = 3
```

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion
- This sometimes requires we define additional parameters that are passed to the method
- Let’s look at an example

Defining Arguments for Recursion

Algorithm reverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

```
if i < j then
    Swap A[i] and A[j]
    reverseArray(A, i + 1, j – 1)
return
```

Here, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

Defining Arguments for Recursion

```
/** Reverses the contents of subarray data[low] through data[high] inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
  if (low < high) {                        // if at least two elements in subarray
    int temp = data[low];                  // swap data[low] and data[high]
    data[low] = data[high];
    data[high] = temp;
    reverseArray(data, low + 1, high − 1); // recur on the rest
  }
}
```

Time Complexity of a Recursive Program

```
public static int factorial(int n) {

        int factorial_value;

        factorial_value = 0;

        /* compute factorial value recursively */

        if (n <= 1) {
                factorial_value = 1;

        }
else {
                factorial_value = n * factorial(n-1);
        }

        return (factorial_value);
}
```

- Let the time complexity of the function be T(n)
- Now, let's try to analyse the algorithm

	n>1
`public static int factorial(int n) {`	
`    int factorial_value;`	1
`    factorial_value = 0;`	1
`    /* compute factorial value recursively */`	
`    if (n <= 1) {`	1
`        factorial_value = 1;`	0
`    }`	
`else {`	1
`        factorial_value = n * factorial(n-1);`	T(n-1)
`    }`	
`    return (factorial_value);`	1
`}`	

Time Complexity

$$T(n) = 5 + T(n-1)$$

$$T(n) = c + T(n-1)$$

$$T(n-1) = c + T(n-2)$$

$$T(n) = c + c + T(n-2)$$
$$= 2c + T(n-2)$$

$$T(n-2) = c + T(n-3)$$

$$T(n) = 2c + c + T(n-3)$$
$$= 3c + T(n-3)$$

$$T(n) = ic + T(n-i)$$

Time Complexity

$T(n) = ic + T(n\text{-}i\ )$

Finally, when $i = n\text{-}1$

$$\begin{aligned} T(n) &= (n\text{-}1)c + T(n\text{-}(n\text{-}1)\ ) \\ &= (n\text{-}1)c + T(1) \\ &= (n\text{-}1)c + d \end{aligned}$$

Hence, $T(n) = \mathrm{O}(n)$

Pitfalls of Recursion

- Recursion can be easily misused

```
/** Returns true if there are no duplicate values from data[low] through data[high].*/
public static boolean unique3(int[ ] data, int low, int high) {
  if (low >= high) return true;                               // at most one item
  else if (!unique3(data, low, high−1)) return false;         // duplicate in first n−1
  else if (!unique3(data, low+1, high)) return false;         // duplicate in last n−1
  else return (data[low] != data[high]);                      // do first and last differ?
}
```

  - Non-recursive part takes O(1)
  - So, T(n) will be proportional to the total number of recursive invocations

$$1 + 2 + 4 + \cdots + 2^{n-1}$$

Pitfalls of Recursion

- Inefficient recursion for computing Fibonacci numbers

```
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
  if (n <= 1)
    return n;
  else
    return fibonacciBad(n−2) + fibonacciBad(n−1);
}
```

Fibonacci using binary recursion

Pitfalls of Recursion

- Inefficient recursion for computing Fibonacci numbers

```
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
  if (n <= 1)
    return n;
  else
    return fibonacciBad(n−2) + fibonacciBad(n−1);
}
```

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$

Pitfalls of Recursion

- Efficient recursion for computing Fibonacci numbers

```
/** Returns array containing the pair of Fibonacci numbers, F(n) and F(n−1). */
public static long[ ] fibonacciGood(int n) {
  if (n <= 1) {
    long[ ] answer = {n, 0};
    return answer;
  } else {
    long[ ] temp = fibonacciGood(n − 1);              // returns {F_{n−1}, F_{n−2}}
    long[ ] answer = {temp[0] + temp[1], temp[0]};    // we want {F_n, F_{n−1}}
    return answer;
  }
}
```

Fibonacci using linear recursion, which caches the partial results as the recursion happens!

Pitfalls of Recursion

- Infinite recursion

```
/** Don't call this (infinite) version. */
public static int fibonacci(int n) {
  return fibonacci(n);
}
```

- ***StackOverflowError:*** Java's safety measure against infinite recursion