Recursion

What is Recursion?

- **Recursion**: is the concept of defining a method that makes a call to itself (**a recursive call**).
- A **recursive method**, is a method that calls itself.

What is Recursion - 2

- Does it do exactly the same thing???
  - → **Logic** (instructions) are exactly the same.
  - → **Conditions** (e.g. parameters, state of data, etc.) are different.

- Like iteration, recursion is a means for repetition
  - Eventually the repetition *must* stop!
  - → The repetitive steps must work towards the stopping case(s).

Classic Example– The Factorial Function

- n! = n * (n-1) * (n-2) * … * 3 * 2 * 1
- Factorial definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & n \geq 1 \end{cases}$$

- What is 4! ?
  - 4! = 4 * 3 * 2 *1
    = 4 * ?

Classic Example– The Factorial Function

- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & n \geq 1 \end{cases}$$

- As a Java method:

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
    if  (n  ==  0)  return  1;
        else return  n  *  recursiveFactorial(n- 1);
}
```

A Closer Look

- As a Java method:

```
// recursive factorial function

public static int  recursiveFactorial(int n) {
    if  (n  ==  0)  return  1;
    else return  n  *  recursiveFactorial(n- 1);
}
```

Base Case(s)

A Closer Look

- As a Java method:

```
// recursive factorial function

public static int  recursiveFactorial(int n) {

    if  (n  ==  0)  return  1;

    else return  n  *  recursiveFactorial(n- 1);

}
```

Base Case(s)

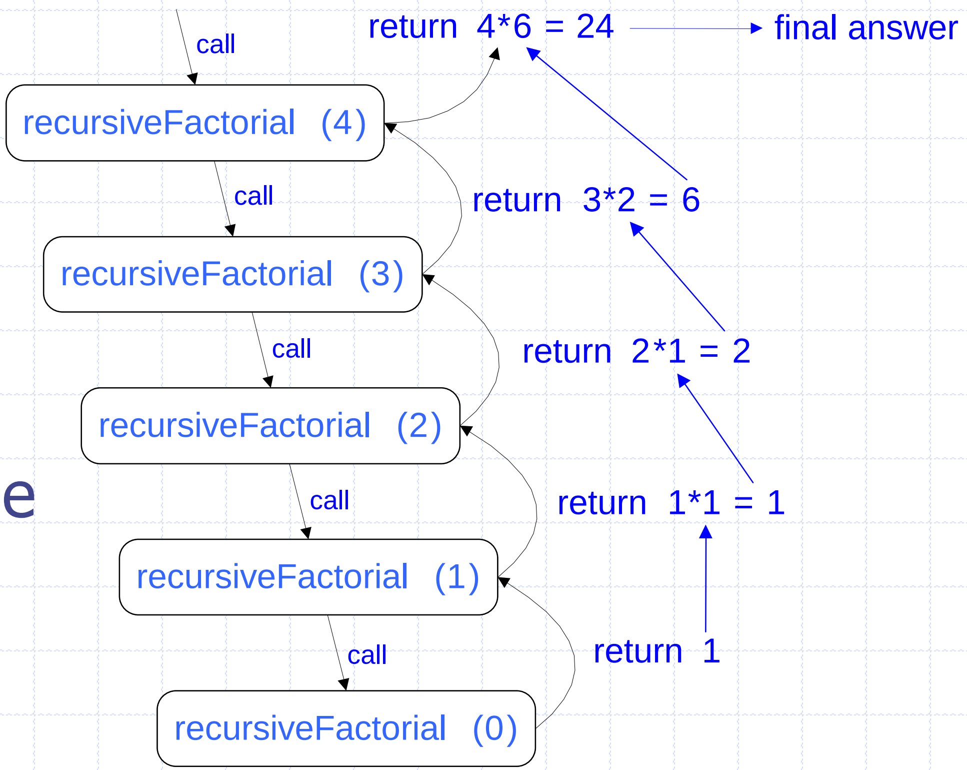
Recursive Case(s)

Content of a Recursive Method

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example



Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

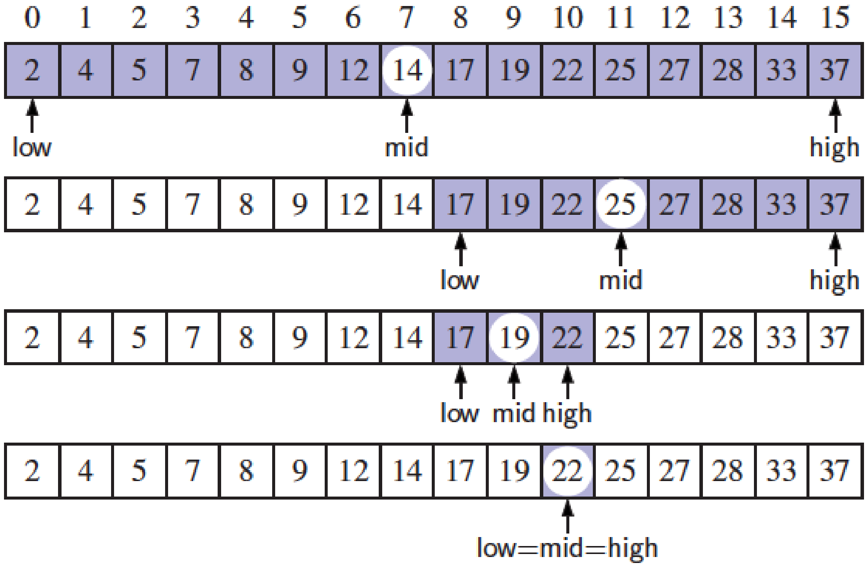
$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

```
public static int  power (int x, int n) {

    if  (n  ==  0)  return  1;
    else return  x  *  power(x, n- 1);
}
```

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Binary Search

Search for an integer in an ordered list

```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                        // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                       // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);   // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);  // recur right of the middle
  }
}
```

Types of Recursion

- Depending on the number of recursive calls performed *each time* a recursive method is invoked there are 3 types:
  1. Linear recursion
     → at most one recursive call.
  2. Binary recursion
     → at most two recursive calls.
  3. Multiple recursion
     → a method may initiate multiple recursive calls

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Reversing an Array

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

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

Example of Linear Recursion

Algorithm linearSum(A, n):

Input:

  Array, A, of integers

  Integer n such that

    $0 \leq n \leq |A|$

Output:

    Sum of the first n integers in A

if n = 0 then

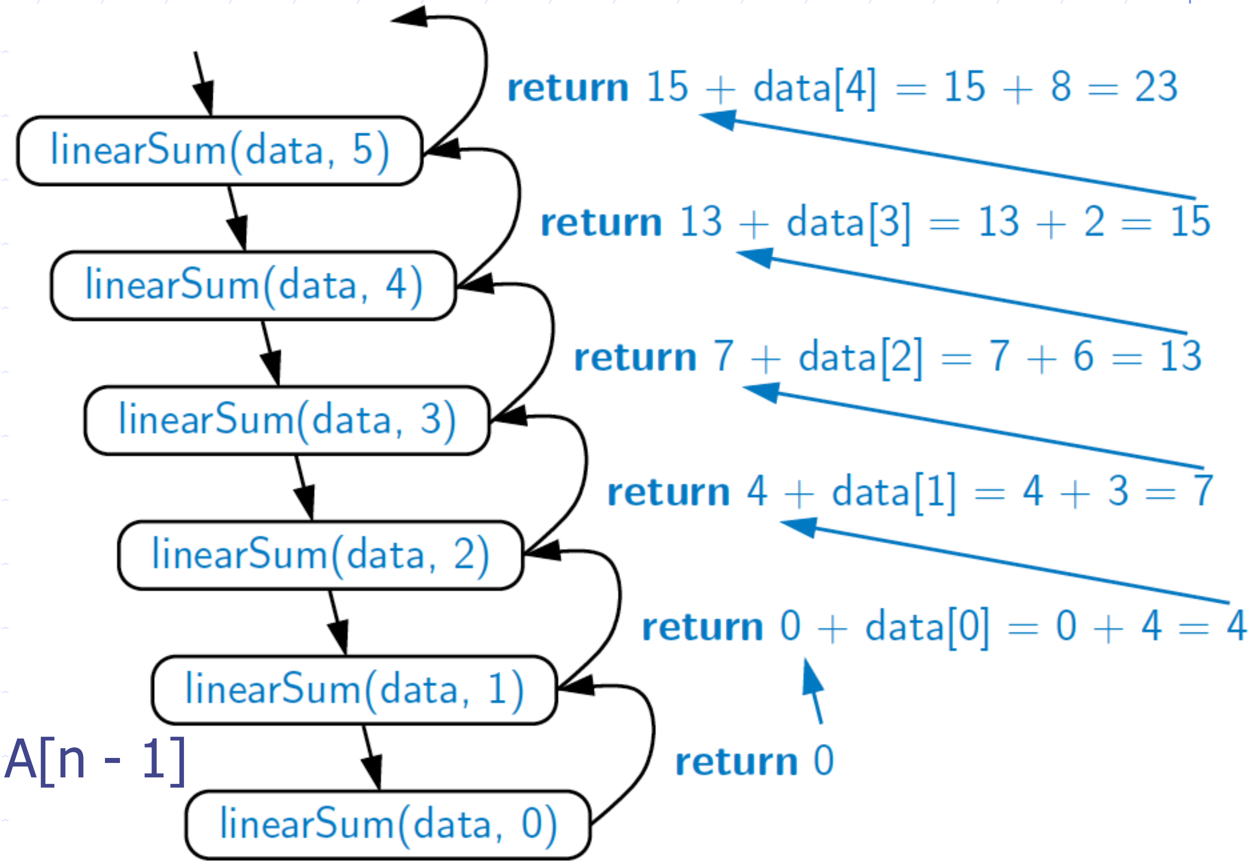
  return 0

else

  return

linearSum(A, n - 1) + A[n - 1]

Recursion trace of linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]



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.
- For example, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

```
/** 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
  }
}
```

Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1.$$

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
  Input: Nonnegative integer k
  Output: The kth Fibonacci number F_k
  if $k <= 1$ **then**
    return k
  else
    return BinaryFib($k - 1$) + BinaryFib($k - 2$)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!