Recursion

Chapter 18

Recursion

- Recursion is an approach to solving problems that breaks a problem into an *identical* but *smaller* problem.
  - You continue breaking the problem into smaller problems until you reach the smallest problem possible.
  - In this smallest problem, the answer is obvious or trivial.
  - You then use this answer to “build back up” and solve the previous problems until you solve the original problem.

- If a problem is easy, solve it now.
- If the problem is hard, solve a small piece of it now, then make it smaller and solve the rest later.

Elements of Recursive Methods

- Execution of a small part of the problem
- A call to the recursive method to finish the remaining problem
- A base case or escape clause- some way to know when the problem is done
- In other words, all recursive methods must:
  - Make the problem smaller
  - Know when to stop

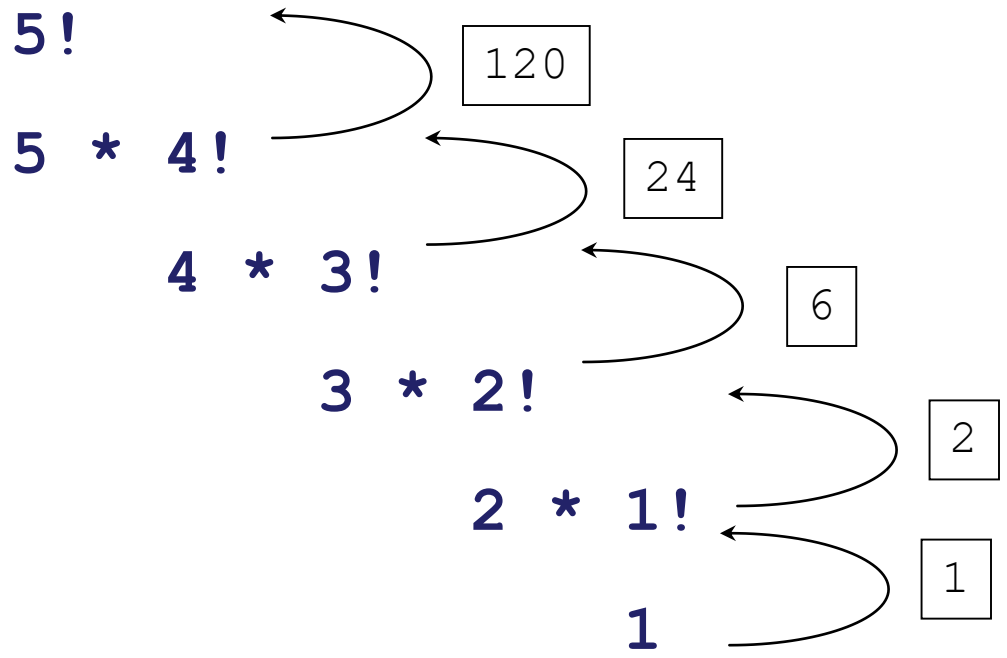
The Base Case

- The non-recursive part of a recursive definition is called the *base case*
- Without a base case, there would be no way to terminate the recursion, creating *infinite recursion*
  - This is similar to an infinite loop
- All recursive definitions must have one or more base cases

Factorial Example

- For any positive integer N, N! is defined as the product of all integers between 1 and N (inclusive)
  - 1! = 1
  - 2! = 2 * 1 = 2
  - 3! = 3 * 2 * 1 = 6
- We can express factorial using recursion:
  - 1! = 1
  - N! = N * (N-1)!
- In this way, a factorial is defined in terms of another factorial
- Eventually, the base case of 1! is reached

Factorial Example (cont.)



Recursive Programming

- A *recursive* method is a method that invokes (calls) itself.
- Recursive methods can be void or can return a value.
- Recursive methods must include:
  - The base case
  - The recursive case- makes the problem smaller!
  - A conditional to determine which case you're in!

Recursive Programming (cont.)

- Each call to the method sets up a new execution environment
  - New parameters
  - New local variables
- As with any other method call, when the method completes, the control returns to the method that invoked it (which might an earlier invocation of itself)
  - Review the example abut activation records and the runtime stack

Example: Summing Integers

- Sum all the numbers between 1 and a positive integer N
- This problem can be defined recursively as:

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i$$

$$= N + N-1 + \sum_{i=1}^{N-2} i$$

$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

$$\vdots$$

- What is the base case?

Practice

- Write this method and trace it when invoked with 5.

Practice- Void Methods

- Write a program that counts down from some number to zero and prints each number.
- Write a method to display the contents of an array.
- Write a method to print the contents of an array backwards.

Recursive Valued Methods

- A critical note about recursive valued methods: you must either:
  a) **return** the value of the recursive method call or
  b) **update** a local variable with the value of the recursive method call (and then return that local variable)
- If you don't do this step, the recursive calls are not linked together and your method will not work!
- This is a very common mistake to make!

Practice- Methods that Return a Value

- Write a recursive method to read input within a specified range and return that input.
- Write a method to return the number of times a character appears in a string.

Practice- Tracing Recursion

```
System.out.println(recMethod1(5, 1));

public int recMethod1(int x, int y) {
    if (x == y)
        return 0;
    else
        return recMethod1(x-1, y) + 1;
}
```

Practice- Tracing Recursion

```
public int recFactorial1(int x) {
    System.out.print(x);
    if (x > 1)
        return x * recFactorial1(x - 1);
    else
        return 1;
}

public int recFactorial2(int x) {
    int fac;
    if (x > 1) {
        fac = x * recFactorial2(x - 1);
        System.out.print(x);
    } else {
        fac = 1;
    }
    return fac;
}
```

Practice- Tracing Recursion

```
int[] a = {3, 2, 1, 2, 3};
System.out.println(recMethod2(a, 2, 0));
System.out.println(recMethod2(a, 2, 2));

public int recMethod2(int[] arr, int b, int c) {
    if (c < arr.length) {
        if (arr[c] != b)
            return recMethod2(arr, b, c + 1);
        else
            return 1 + recMethod2(arr, b, c + 1);
    } else {
        return 0;
    }
}
```

Recursion and Iteration

- Any problem that can be solved with recursion can be solved with iteration.
- And vice versa.

Recursion vs. Iteration

- Just because you *can* use recursion to solve a problem, doesn't mean you *should*
- For example, the summing 1 to N problem could be implemented easily with iteration

```
int result = 0;
for(int i=1; i<=N; i++)
   result += i;
```

- However, for some problems, recursion provides a solution that is easier to understand

Recursion vs. Iteration (cont.)

- Whether to use recursion or iteration is an important design decision
- Things to consider:
  - How clear is the solution?
  - How easy is the solution to program and test?
  - Is the solution re-using information in the best way?
  - What is the efficiency of the solution?

Example: Fibonacci Sequence

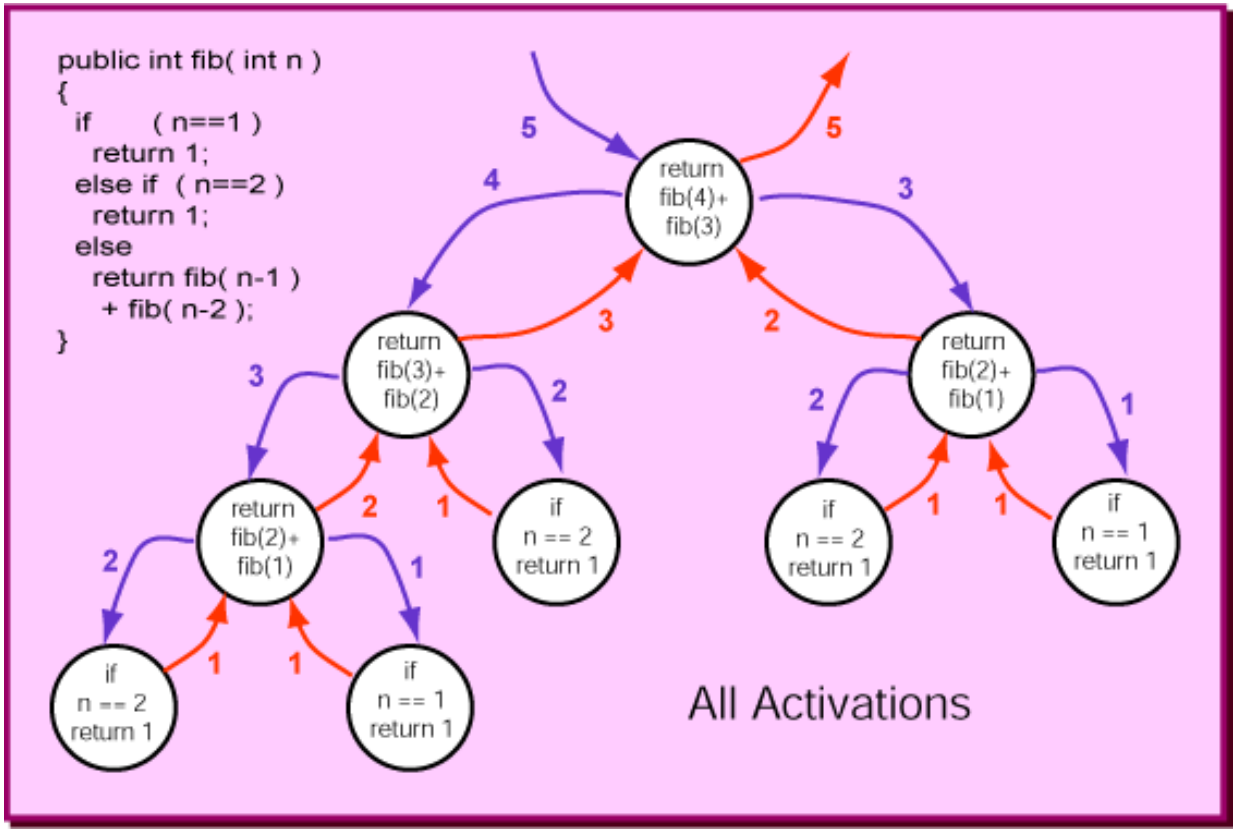
$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1\,. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- The Fibonacci Sequence is used in many areas of math, computer science, and is seen in nature.
- We could program this either recursively or iteratively.

Fibonacci: Recursive Solution

```
public int fibonacciRecusive(int n) {
  if (n < 2) {
     return n;
  } else {
     return fibonacciRecusive(n - 1) +
            fibonacciRecusive(n - 2);
  }
}
```

Fibonacci: Recursive Call Trace



Fibonacci: Iterative Solution

```
public static int fibonacciIterative (int n) {
  int sum1 = 0, sum2 = 1;

  for(int i=0; i<n; i++) {
      int temp= sum1;
      sum1 = sum2;
      sum2 = temp + sum2;
  }
  return sum1;
}
```

Example: Fibonacci Sequence (cont.)

- The recursive solution is easier to understand and is based on the mathematical definition of the algorithm.
- However, the recursive solution is $O(2^n)$ and re-calculates things many times.
- Iterative solution is linear, it will only execute n times.
- The iterative solution is much more efficient.