Recursion

A First Look

Dean Hendrix

Recursion

- **Recursion** is a means of specifying the solution to a problem in terms of solutions to smaller instances of the same problem.
- The *smallest* instance of the problem must have a solution that is known or trivial to compute; that is, one that does not involve recursion.

- We were first taught to solve problems *iteratively* (i.e., with loops), but a recursive solution is often just as natural and sometimes moreso.
- Iteration and recursion are equally powerful, though. Any solvable problem can be solved with either technique.
- We will develop the basic idea of recursion through computing a simple mathematical function on positive integers — factorial.

Factorial

The **factorial** of a positive integer n, denoted $n!$, is the product of all the positive integers less than or equal to n.

$$n! = \prod_{i=1}^{n} i$$

- Example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Factorial

We can compute the factorial function with **iteration** like so:

```
public int factorial(int n) {
    int fact = n;
    for (int i = n - 1; i > 0; i--) {
        fact = fact * i;
    }
    return fact;
}
```

A trace of `factorial(5)` :

fact	i
5	4
20	3
60	2
120	1
120	0

Factorial

Let's look again at the expansion of $5! = \prod_{i=1}^{n} i$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Observation: We can express the factorial of any integer in terms of the factorial of smaller integers.

$$5! = 5 \times \underline{4 \times 3 \times 2 \times 1} = 5 \times 4!$$

And this holds true at every level:

$5! = 5 \times 4!$
$5! = 5 \times 4 \times 3!$
$5! = 5 \times 4 \times 3 \times 2!$
$5! = 5 \times 4 \times 3 \times 2 \times 1!$
$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Recursion

- **Recursion** is a means of specifying the solution to a problem in terms of solutions to smaller instances of the same problem.
- The *smallest* instance of the problem must have a solution that is known or trivial to compute; that is, one that does not involve recursion.

Factorial

Since we now see the recursive structure of the factorial function, we can express its definition recursively.

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{if } n > 1 \end{cases}$$

Note that this definition has two parts:

1. A solution to the smallest instance of the problem.
   - This is called the **base case**.
2. A rule for reducing all other instances to the base case.
   - This is called the **recursive step** or the **reduction step**.

Factorial

We can compute the factorial function with **recursion** like so:

```java
public int factorial(int n) {
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

A trace of `factorial(5)` :

```
factorial(5)
    5 * factorial(4)
        4 * factorial(3)
            3 * factorial(2)
                2 * factorial(1)
                    1
                2 * 1
            3 * 2
        4 * 6
    5 * 24
120
```

Factorial

The iterative version:

```
public int factorial(int n) {
    int fact = n;
    for (int i = n - 1; i > 0; i--) {
        fact = fact * i;
    }
    return fact;
}
```

The recursive version:

```
public int factorial(int n) {
    if (n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

Search

Let's revisit a familiar problem and develop both an iterative solution and a recursive one.

```
/**
 * Returns true if target is in a[start]..a[a.length-1],
* false otherwise.
*/
public boolean search(int[] a, int target, int start)
```

Example calls:

a	target	start	return value
{2,4,6,8,10}	4	0	true
{2,4,6,8,10}	4	2	false

Search

An iterative solution:

```
public boolean search(int[] a, int target, int start) {
    for (int i = start; i < a.length; i++) {
        if (a[i] == target) {
            return true;
        }
    }
    return false;
}
```

Search

```
public boolean search(int[] a, int target, int start) {
    for (int i = start; i < a.length; i++) {
        if (a[i] == target) {
            return true;
        }
    }
    return false;
}
```

A trace of `search({2,4,6,8,10}, 4, 2)`:

`i`	$i < a.length$	`a[i] == target`
2	true ($2 < 5$)	false ($6 \neq 4$)
3	true ($3 < 5$)	false ($8 \neq 4$)
4	true ($4 < 5$)	false ($10 \neq 4$)
5	false ($5 = 5$)	
return false		

Recursion

- **Recursion** is a means of specifying the solution to a problem in terms of solutions to smaller instances of the same problem.
- The *smallest* instance of the problem must have a solution that is known or trivial to compute; that is, one that does not involve recursion.

- The smallest instance of this problem?
  - An "empty" search area. That is, `start` is after the last index.
- The "general" instance of this problem?
  - A search area with at least one element.
  - If `a[start]` is what we're looking for we can return true.
  - Otherwise, we need to return the result of searching starting with the next index.

A recursive solution:

```
public boolean search(int[] a, int target, int start) {
    if (start == a.length) {
        return false;
    }
    if (a[start] == target) {
        return true;
    }
    return search(a, target, start + 1);
}
```

Search

```
public boolean search(int[] a, int target, int start) {
    if (start == a.length) {
        return false;
    }
    if (a[start] == target) {
        return true;
    }
    return search(a, target, start + 1);
}
```

A trace of `search({2,4,6,8,10}, 4, 2)`:

`a[start..a.length-1]`	`target`	`start`	`start == a.length`	`a[start] == target`
{6,8,10}	4	2	false ($2 \neq 5$)	false ($6 \neq 4$)
{8,10}	4	3	false ($3 \neq 5$)	false ($8 \neq 4$)
{10}	4	4	false ($4 \neq 5$)	false ($10 \neq 4$)
{}	4	5	true ($5 = 5$)	
return false				