Recursion

A programming technique in which a method calls itself

Alwin Tareen

Recursion

A description of the recursive technique

- Recursion is a programming techique in which a method calls itself.
- Recursion can always be used in place of iteration, and iteration can always be used in place of recursion.
- There are many situations in which recursion provides the clearest, shortest and most elegant solution to a programming task.

A recursive method has two kinds of cases:

- One or more stopping or base cases that solve the problem without any recursive calls.
- One or more cases that include a recursive call(involving a simpler problem).

Guidelines for Writing Recursive Methods

Must have a base case

- Just as we guard against writing infinite loops, we must avoid recursions that never come to an end.
- A recursive method must have a well-defined termination or stopping state, also referred to as the **base case**.
- For example:

```
if (n == 1)
{
    return 1;
}
```

Guidelines for Writing Recursive Methods

The recursive case must approach the base case

- The recursive step, in which the method calls itself, must eventually lead to a base case.
- Since each invocation of the method is passed a smaller value, eventually the stopping state must be reached.
- If a method failed to reach the stopping state, the `Java` interpreter would run out of memory, at which point the program would terminate with a `StackOverflow` error.
- For example:

```
else
{
   return n * factorial(n-1);
}
```

Characteristics of Recursive Methods

The following are some key features common to all recursive routines:

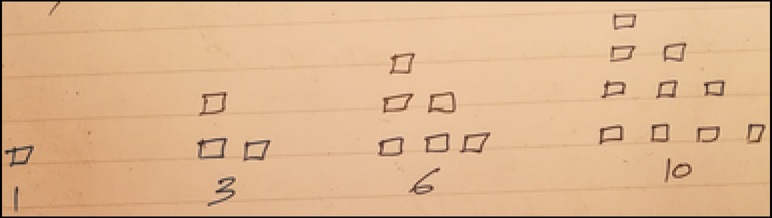
- The method calls itself.
- When the method calls itself, if does so to solve a smaller problem.
- There's some version of the problem which is so simple that the method can solve it, and return. This is the base case.

Triangular Numbers

- Starting with `1`, the `nth` term in a triangular series is obtained by adding `n` to the previous term.

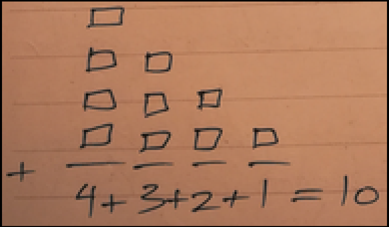
element	1	3	6	10	15	21
n	1	2	3	4	5	6

- These numbers can be visually arranged as a triangular arrangement of objects:



Triangular Numbers

- Say you wanted to calculate the `n`th term in a triangular series.
- You may decide that the value of any term can be obtained by adding up all of the vertical columns of squares.



Finding Triangular Numbers Iteratively

- The following program uses this column-based technique to find a triangular number.
- The method cycles around the loop `n` times, adding `n` to the total the first time, `n-1` the second time, and so on down to 1, quitting the loop when `n` becomes 0.

Finding Triangular Numbers Iteratively

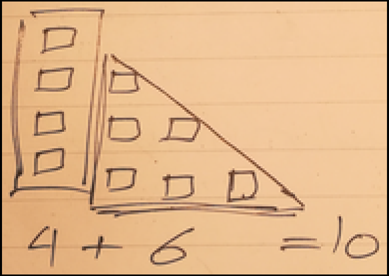
```
public class TriangularIterative
{
    public static int triangular(int n)
    {
        int total = 0;
        while (n > 0)
        {
            total = total + n;
            n--;
        }
        return total;
    }

    public static void main(String[] args)
    {
        int result = triangular(4);
        System.out.println("n = 4, triangular = " + result);
    }
}
```

Finding Triangular Numbers Recursively

The value of the `n`th term can be thought of as the sum of only two things:

1. The first(tallest) column, which has the value `n`.
2. The sum of all the remaining columns.



Finding Triangular Numbers Recursively

- The sum of all the remaining columns for term `n` is the same as the sum of all the columns for term `n-1`.
- Therefore, all we have to do is call the `triangular()` method again, but with an argument of `n-1`.
- We must also provide a condition that leads to a recursive method returning, without making another recursive call. This is the base case.
- It's critical that every recursive method has a base case to prevent infinite recursion.

Finding Triangular Numbers Recursively

```
public class TriangularRecursive
{
    public static int triangular(int n)
    {
        if (n == 1)
        {
            return 1;
        }
        else
        {
            return n + triangular(n-1);
        }
    }

    public static void main(String[] args)
    {
        int result = triangular(4);
        System.out.println("n = 4, triangular = " + result);
    }
}
```

Factorial Numbers

- Factorial numbers are similar in concept to triangular numbers, except that multiplication is used instead of addition.
- The factorial of `n` is found by multiplying `n` by the factorial of `n-1`.

factorial	1	1	2	6	24	120	720
n	0	1	2	3	4	5	6

- Note that the factorial of `0` is defined to be `1`.

Factorial Numbers

- There are only two differences between the recursive program for triangular numbers, and the recursive program for factorial numbers.
- In the factorial program, the numbers are being multiplied together, not added.
- In the factorial program, the base condition occurs when `n` is 0, instead of 1.

The definition of the factorial function

- The definition of factorial can be written recursively.
- This means that the factorial of the number `n` can use the factorial of the previous number, `n-1`, in its computation.
- $n! = n * (n-1)!$

Finding Triangular Numbers Recursively

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

    public static void main(String[] args)
    {
        int result = factorial(5);
        System.out.println("n = 5, factorial = " + result);
    }
}
```

Fibonacci Numbers

- In the fibonacci sequence, the first two terms are 1, and then each term starting with the third term is equal to the sum of the previous two terms.

fibonacci	1	1	2	3	5	8	13	21
n	1	2	3	4	5	6	7	8

Note that the recursive implementation of fibonacci requires the following:

- Two base cases, for when `n=1` and `n=2`.
- Two recursive calls to `fibonacci()`.

Finding Fibonacci Numbers Recursively

```
public class FibonacciRecursive
{
    public static int fibonacci(int n)
    {
        if (n == 1)
        {
            return 1;
        }
        else if (n == 2)
        {
            return 1;
        }
        else
        {
            return fibonacci(n-1) + fibonacci(n-2);
        }
    }
```

Finding Fibonacci Numbers Recursively, Continued

```
    public static void main(String[] args)
    {
        int result = fibonacci(6);
        System.out.println("n = 6, fibonacci = " + result);
    }
}
```

Recursion: End of Notes