Recursion

Authored by AllisonP

Table Of Contents

Recursion
Example (Java)

Recursion

This is an extremely important algorithmic concept that involves splitting a problem into two parts: a *base case* and a *recursive case*. The problem is divided into smaller subproblems which are then solved recursively until such time as they are small enough and meet some base case; once the base case is met, the solutions for each subproblem are combined and their result is the answer to the entire problem.

If the base case is not met, the function's recursive case calls the function again with modified values. The code must be structured in such a way that the base case is reachable after some number of iterations, meaning that each subsequent modified value should bring you closer and closer to the base case; otherwise, you'll be stuck in the dreaded infinite loop!

It's important to note that any task that can be accomplished recursively can also be performed iteratively (i.e., through a sequence of repeatable steps). Recursive solutions tend to be easier to read and understand than iterative ones, but there are often performance drawbacks associated with recursive solutions that you're going to want to evaluate on a case-by-case basis. Typically, we use recursion when each recursive call significantly reduces the size of the problem (e.g., if we can halve the dataset during each recursive call). Regardless of the advisability of recursively solving a problem, it's extremely important to practice and understand *how* to recursively solve problems.

Example (Java)

\- EXAMPLE

The code below produces the multiple of two numbers by combining addition and recursion.

Input Format
Two space-separated integers to be multiplied.

```java
import java.util.*;

class Solution {
    // Multiply 'n' by 'k' using addition:
    private static int nTimesK(int n, int k) {
        // Print current value of n
        System.out.println("n: " + n);

        // Recursive Case
        if(n > 1) {
            return k + nTimesK(n - 1, k);
        }
        // Base Case n = 1
        else {
            return k;
        }
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int result = nTimesK(scanner.nextInt(), scanner.nextInt());
        scanner.close();
        System.out.println("Result: " + result);
    }
}
```

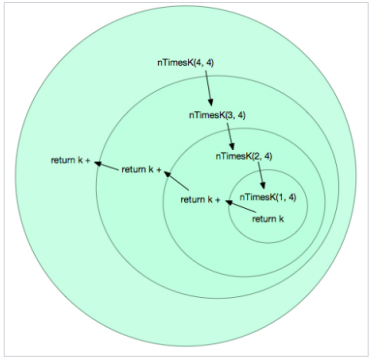
Input

```
4 4
```

Run

Output

The diagram below depicts the execution of the code above using the default input (`4 4`). Each call to *nTimesK* is represented by a bubble, and each new recursive call bubble is stacked inside and on top of the bubble that was responsible for calling it. The function recursively calls itself using reduced values until it reaches the base case ($n = 1$). Once it reaches the base case, it passes back the base case's return value ($k = 4$) to the bubble that called it and continues passing back `k +` the previously returned value until the final result (i.e.: the multiplication by addition result of $n \times k$) is returned.



Once the code hits the base case in the 4^{th} bubble, it returns k (which is 4) to the 3^{rd} bubble.
Then the 3^{rd} bubble returns $k + 4$, which is 8, to the 2^{nd} bubble.
Then the 2^{nd} bubble returns $k + 8$, which is 12, to the 1^{st} bubble.
Then the 1^{st} bubble returns $k + 12$, which is 16, to the first line in *main* as the result for $nTimesK(4, 4)$, which assigns 16 to the $result$ variable.