TASK

Recursion

Visit our website

Introduction

Welcome to the Recursion Task!

Recursion is a very useful programming tool, that, in many cases, enables you to develop a straightforward, simple solution to an otherwise complex problem. However, it is often difficult to determine how a program can be approached recursively. In this task we explain the basic concepts of recursive programming and teach you to "think recursively".

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

What is Recursion?

When you are faced with a particularly difficult or complex problem it is often easier to break the problem down into smaller, more manageable chunks which are easier to solve. This is the basic idea behind recursion; recursive algorithms break down a problem into smaller pieces that you already know how to solve.

In simple terms, recursion involves having a function call itself until some specified condition has been met. This is similar to a loop, however unlike loops, which work by explicitly specifying a repetition structure, recursion uses continuous function calls to achieve repetition.

Recursion is a somewhat advanced topic and problems that can be solved with recursion, can also most likely be solved by using simpler looping structures. However, recursion is a useful programming technique that, in some cases, can enable you to develop natural, straightforward, simple solutions to otherwise difficult problems.

Recursive Functions

As mentioned previously, a recursive function is a function that calls itself. For example, let's say that you have a cake that you wish to share equally amongst several friends. To do so you might start of by cutting the cake in half and then again cutting the resulting slices in half until there are enough slices for everyone. The code to implement such an algorithm might look something like this:

```
public static int cut_cake(int number_of_friends, int number_of_slices)
{
  // Cut cake in half
  number_of_slices = number_of_slices * 2;

  // Check if there are enough slices for everybody
  if (number_of_slices >= number_of_friends) {
    // If there are enough slices - return the number of slices
    return number_of_slices;
  }
  else {
    // If there are not enough slices - cut the resulting
    // slices in half again.
    return cut_cake(number_of_friends, number_of_slices);
  }
}

public static void main(String[] args) {
  System.out.println(cut_cake(11, 1));
}
```

The cut_cake function takes the number of friends you wish to share the cake with (11) and the number of slices of cake (initially 1 since the cake is not cut). Line 4, number_of_slices = number_of_slices * 2, cuts the cake in half. Line 7 then checks if there are enough slices. If there are enough slices, the number of slices are returned (line 9) and if not the function calls itself again (line 14) in order to cut the cake in half one more time. This is an example of a recursive function.

Computing Factorials using Recursion

Many mathematical functions can be defined using recursion. A simple example is the **factorial function.** The factorial function, n!, describes the operation of multiplying a number by all positive integers less than or equal to itself. For example, 4! = 4*3*2*1

If you look closely at the examples above you might notice that 4! can also be written as 4!=4*3! and 3! can then in turn be written as 3! = 3*2! and so on. The factorial of a number n can therefore be recursively defined as follows:

```
0! = 1
n! = n × (n - 1)! where n > 0
```

Assuming that you know (n - 1)!, you can easily obtain n! immediately by using n × (n - 1)!. The problem of computing n! Is therefore reduced to computing (n - 1)!. When computing (n - 1)!, you can apply the same idea recursively until n is reduced to 0. The recursive function for calculating n! Is shown below:

```
public static long factorial(int n)
{
    // Base case: 0! = 1
    if (n == 0)
        return 1;
    else
        // Recursive case: n! = n * (n-1)!
        return n * factorial(n-1);
}
```

If you call the function factorial(n) with n = 0, it immediately returns a result of 1. This is known as the **base case** or the stopping condition. The base case of a function is the problem that we already know the answer to. In other words it can be solved without any more recursive calls. The base case is what stops the recursion from continuing on forever. Every recursive function must have at least one base case.

If you call the function factorial(n) with n > 0, the function reduces the problem into a subproblem for computing the factorial of n - 1. The subproblem is essentially a simpler or smaller version of the original version. Because the subproblem is the same as the original problem, you can call the function again, this time with n - 1 as the argument. This is referred to as a recursive call. A **recursive call** can result in many more recursive calls, because the function keeps on dividing a subproblem into new subproblems. For a recursive function to terminate, the problem must eventually be reduced to a base case.

In general there are two main requirements for a recursive function:

- **Base case:** the function returns a value when a certain condition is satisfied, without any other recursive calls.
- **Recursive call:** the function calls itself with an input which is a step closer to the base case.

Compulsory Task

Follow these steps:

- In a file called “recursion.java”, create:
    - a recursive function that reverses a string
    - a recursive function that, given a number n, prints out the first n Fibonacci numbers (Fibonacci numbers are a sequence where each number is the sum of the previous two - 0 1 1 2 3 5 8...)

Optional Task

Follow these steps:

- Create a program that implements a search and replace function recursively. Your program should allow a user to enter a string, a substring they wish to find and another string they wish to replace the found substring with.

  The output of your program should be similar to the output given below:

```
Please enter a string: Hello world
Please enter the substring you wish to find: llo
Please enter a string to replace the given substring: @@
Your new string is: he@@ world
```

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.