TASK

Applied Recursion

Visit our website

Introduction

WELCOME TO THE APPLIED RECURSION TASK!

Recursion is a handy 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 an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team 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 faced with a particularly difficult or complex problem, it is often easier to break the problem down into smaller, more manageable chunks that 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 is when a function calls itself. Normally a recursive function uses conditional statements to determine whether or not to call the function recursively. The main benefits of using recursion are compactness of code, understandability of code, and having fewer variables. Recursion and iteration (loops) can be used to achieve the same results. 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.

The following guidelines will help you to decide which method to use depending on a given situation:

- **When to use recursion?** When a compact, understandable, and intuitive code is required.
- **When to use iteration?** When there is limited memory and faster processing is required.

We can try to visualise recursion with the Russian doll example. In the image above the dolls are created in a manner where each doll can contain a smaller doll. This repetition, or recursion, cannot continue indefinitely and at some point the creation of a doll that fits inside the smallest doll becomes impossible. The condition of the doll no longer being able to contain a smaller doll is what is known in recursion as the base case.

A recursive function, similarly, follows this pattern. It calls itself within its own code until a specified condition is met (the base case) at which point the recursion ends and a result from the computation is returned.

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 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:

```
function cutCake(numbFriends, numbSlice){
      if(numbFriends <= numbSlice){ /*Base case. This is the condition
that terminates the recursion */
            return numbSlice;
```

```
      } else {
            /*Recursive call. This function doubles the number of cake
slices in its recursion call which will end up making the base case true
and terminates the function.*/
            return cutCake(numbFriends, numbSlice * 2);/*Doubling the
number of slices by cutting the cake into half.*/
      }
}

console.log(cutCake(11, 1));
```

The cutCake function takes the number of friends you wish to share the cake with (11) and the number of slices of cake (initially 1 when the function is called in the print statement since the cake is not yet cut). Line 4, numSlice * 2, cuts the cake in half. Line 2 then checks if there are enough slices. If there are enough slices, the number of slices is returned (line 3) and if not the function calls itself again (line 4) to cut the cake in half again, this time passing the new number of slices after cutting the cake. This is an example of a recursive function.

Current function call	**Action taken after current function call**
1st function call cutCake(11, 1)	New numSlice value passed as an argument in the next recursive function call = 1*2 = 2, together with numFriends (which is 11, because the number of friends stays constant)
2nd function call because numSlices < numFriends after previous function call cutCake(11, 2)	New numSlices value passed as an argument in the next recursive function call = 4 together with numFriends (which is 11, because the number of friends stays constant)
3rd function call because numSlices < numFriends after previous function call cutCake(11, 4)	New numSlices value passed as an argument in the next recursive function call = 8 together with numFriends (which is 11, because the number of friends stays constant)
4th function call because numSlices < numFriends after previous function call cutCake(11, 16)	New numSlices = 16 returned because the base case is met i.e. the number of slices is now greater than the number of friends. The recursion has now ended and the value that is printed is 16.

COMPUTING FACTORIALS USE RECURSION

Many mathematical functions can be defined using recursion. A simple example is a **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!. In turn, 3! can be written as 3! = 3*2! and so on. Therefore, the factorial of a number n can 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! 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:

```
function factorial(n){
      if(n <= 0){ /*Base case. When the value of n is equal to or less
than 0, then 1 will be returned. */
            return 1;
      } else{
            /*recursive call to the factorial function working towards
the base case.*/
            return n * factorial(n - 1);
      }
}

console.log(factorial(20));
```

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 to which we already know the answer. In other words, it can be solved without any more recursive calls. The base case is what stops the recursion from continuing 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 summary, 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 that is a step closer to the base case.

Let's create a trace table for the n! factorial calculation 4!, using the code given above:

Iteration	n	n<=0	n-1
1	4	False	3
2	3	False	2
3	2	False	1
4	1	False	0
5	0	True - base case, so stop recursive calls and return the value for 0!, which is 1.	

When the base case is reached, you gain a value to calculate with, the easy and known case of 0! = 1. The 1 is passed back to the previous calling function, giving 1*1=1; this is passed back to the previous calling function giving 1*2=2; this is passed back to the previous calling function giving 2*3= 6; and finally, this is passed back to the very first calling function, giving 6*4=24. The function then terminates and the result, 24, is output via `console.log`.

You could check to confirm that this is what is actually happening by stepping through the code one instruction at a time if your coding environment allows this, or by adding a counter that increments every time the code iterates recursively.

Compulsory Task 1

Follow the following steps.

- Create a file called **reverseSent.js**. In this file:
    - Create a function called reverseSent that takes two arguments, which are the sentence and the position.
    - This function should use recursion to reverse a string sentence and output the results to the console.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Compulsory Task 2

Follow the following steps:

- Create a file called **searchIndex.js** and in this file create a function which:
    - will recursively search for a word in an array and
        - If the word is found, return the index position of that word
        - If the word is not found, return -1 if the word is not found.

Example:

- Consider an array with the following words:

dataArray = ["java", "html", "javascripts", "css"];

- If the word “html” is passed as the word to be searched for, the function will return 1, since this word is found on index 1, the second position of the array (remember that arrays are numbered from 0).

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Optional Bonus Task

Follow these steps:

- Create a file called **GCDRecursion.js**, and in this file create:
  - a recursive function to calculate the greatest common divisor (**GCD**) for two numbers x and y, using recursion.
- Keep in mind that the GCD of two or more integers is the largest positive integer that divides each of those two or more integers without leaving a remainder. For example, the GCD of 5 and 10 is 5.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

Rate us

Share your thoughts

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

Think that 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.