Recursive Functions

A definition that defines an object in terms of itself is said to be *recursive*. In computer science, recursion refers to a function or subroutine that calls itself, and it is a fundamental paradigm in programming. A recursive program is used for solving problems that can be broken down into sub-problems of the same type, doing so until the problem is easy enough to solve directly.

Contents

Examples

Fibonacci Numbers

A common recursive function that you've probably encountered is the Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, and so on. That is, you get the next Fibonacci number by adding together the previous two. Mathematically, this is written as

$$f(N) = f(N-1) + f(N-2)$$

Try finding f(10). No doubt, you have the correct answer, because you intuitively stopped when you reach $f(1)$ and $f(0)$. To be formal about this, we need to define when the recursion stops, called the *base cases*. The base cases for the Fibonacci function is $f(0) = 0$, and $f(1) = 1$. The typical way to write this function is as follows:

$$f(N) = \begin{cases} N & \text{if } N \le 1 \\ f(N-1) + f(N-2) & \text{if } N > 1 \end{cases}$$

Here is a Python implementation of the Fibonacci function:

```
def Fibonacci(x):
  if (x <= 1) return x
  return Fibonacci(x-1) + Fibonacci(x-2)
```

(As a challenge to the reader: How could you implement the Fibonacci function without using recursion?)

Factorial Function

Consider the factorial function, $n! = n * (n-1) * \ldots * 1$, with 0! defined as having a value of 1. We can define this recursively as follows:

$$f(x) = \begin{cases} 1 & \text{if } x = 0 \\ x * f(x-1) & \text{if } x > 0 \end{cases}$$

With this definition, the factorial of every non-negative integer can be found.

Here is a Python implementation of the factorial function:

```
def Factorial(x):
  if (x == 0) return 1
  return x * Factorial(x-1)
```

Some Definitions

A few definitions: *Indirection recursion* is when a function calls another function which eventually calls the original function. For example, A calls B, and then, before function B exits, function A is called (either by B or by a function that B calls). *Single recursion* is recursion with a single reference to itself, such as the factorial example above. *Multiple recursion*, illustrated by the Fibonacci number function, is when a function has multiple self references. *Infinite recursion* is a recursive function that never returns because it keeps calling itself. The program will eventually crash with an *out of memory* error message of some sort.

This ACSL category focuses on mathematical recursive functions rather than programming algorithms. While many mathematical functions can be done iteratively more efficiently than they can be done recursively, many algorithms in computer science must be written recursively. The Tower of Hanoi which is referred to in one of the videos is a good example of this. Another example is given in the sample problems below.

Sample Problems

This ACSL category focuses on mathematical recursive functions rather than programming procedures; but you'll see some of the latter. You will typically be asked to evaluate a recursive function for some specific value. It's important that you are careful and neat. Work your way down the recursive calls until there are no more calls, and then work your way back up.

Sample Problem 1

Problem: Find $g(11)$ given the following

$$g(x) = \begin{cases} g(x-3)+1 & \text{if } x > 0 \\ 3x & \text{otherwise} \end{cases}$$

Solution:

The evaluation proceeds top-down as follows:

$$\begin{aligned} g(11) &= g(8)+1 \\ g(8) &= g(5)+1 \\ g(5) &= g(2)+1 \\ g(2) &= g(-1)+1 \\ g(-1) &= -3 \end{aligned}$$

We now use what we know about $g(-1)$ to learn more values, working our way back "up" the recursion:

We now know the value of $g(2) : g(2) = -3 + 1 = -2$
We now know the value of $g(5) : g(5) = -2 + 1 = -1$
We now know the value of $g(8) : g(8) = -1 + 1 = 0$
And finally, we now have the value of $g(11) : g(11) = 0 + 1 = 1$

Sample Problem 2

Problem: Find the value of $h(13)$ given the following definition of h:

$$h(x) = \begin{cases} h(x-7)+1 & \text{when } x > 5 \\ x & \text{when } 0 \le x \le 5 \\ h(x+3) & \text{when } x < 0 \end{cases}$$

Solution:

$$\begin{aligned} h(13) &= h(6)+1 && \text{top rule, since } x > 5 \\ h(6) &= h(-1)+1 && \text{top rule, since } x > 5 \\ h(-1) &= h(-1+3) = h(2) && \text{bottom rule, since } x < 0 \\ h(2) &= 2 && \text{middle rule, since } 0 \le x \le 5 \end{aligned}$$

We now work our way back up the recursion, filling in values that have been computed.

$$h(-1) = h(2) = 2$$
$$h(6) = h(-1) + 1 = 2 + 1 = 3$$
$$h(13) = h(6) + 1 = 3 + 1 = 4$$

Sample Problem 3

Problem: Find the value of $f(12,6)$ given the following definition of f:

$$f(x,y) = \begin{cases} f(x-y, y-1) + 2 & \text{when } x > y \\ x + y & \text{otherwise} \end{cases}$$

Solution:

$$\begin{aligned} f(12,6) &= f(6,5) + 2 && \text{top rule, since } x > y \\ f(6,5) &= f(1,4) + 2 && \text{top rule, since } x > y \\ f(1,4) &= 1 + 4 = 5 && \text{bottom rule, since } x \le y \end{aligned}$$

We now work our way back up the recursion, filling in values that have been computed.

$$f(12,6) = f(6,5) + 2 = f(1,4) + 2 + 2 = 5 + 2 + 2 = 9$$

Sample Problem 4

Problem: Consider the following recursive algorithm for painting a square:

1. Given a square. 2. If the length of a side is less than 2 feet, then stop. 3. Divide the square into 4 equal size squares (i.e., draw a “plus” sign inside the square). 4. Paint one of these 4 small squares. 5. Repeat this procedure (start at step 1) for each of the 3 unpainted squares.

If this algorithm is applied to a square with a side of 16 feet (having a total area of 256 sq. feet), how many square feet will be painted?

Solution:

In the first pass, we get four squares of side 8. One is painted, three are unpainted. Next, we have 3*4 squares of side 4. Three are painted (area=$3*4^2$), nine are not. Next, we have 9*4 squares of side 2. Nine are painted (area = $9*2^2$), 27 are not. Finally, we have 27*4 squares of side 1. Twenty-seven are painted. Therefore, the total painted is $1*8^2 + 3*4^2 + 9*2^2 + 27*1^2 = 175$.

Online Resources

ACSL

The following videos show the solution to problems that have appeared in previous ACSL contests.

ACSL Math: Recursive Functions

"ACSL Math: Recursive Functions" (Quick Coding Bytes) (https://youtu.be/YinptBVZNFM)

This video introduces the topic, then using an example problem, explains the methodology to solve problems that appear on ACSL contests.

Recursion Example 1

Recursion Example 1 (**CalculusNguyenify**) (https://youtu.be/OjZSIXStSr8)

The video walks through the solution to a straight-forward single-variable recursive function, that is, $f(x) = \{ \ldots$. The problem appeared in ACSL Senior Division Contest #1, 2014-2015.

Recursion Example 2

Recursion Example 2 (**CalculusNguyenify**) (https://youtu.be/MWdGTVCLI8g)

The video walks through the solution to a 2-variable recursive function, that is, $f(x, y) = \{ \ldots$. The problem appeared in ACSL Senior Division Contest #1, 2014-2015.

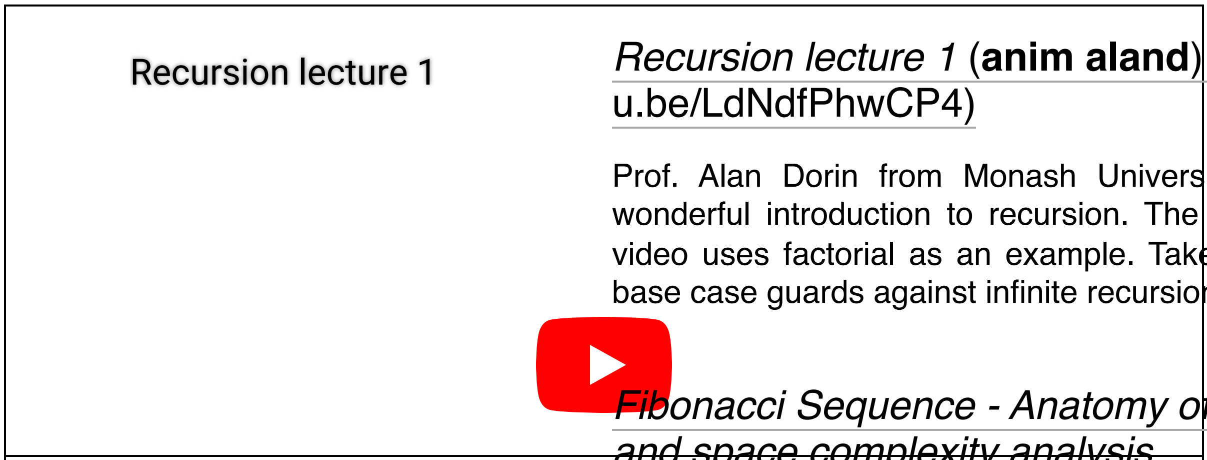
Recursive Functions ACSL Example Problem

Recursive Functions ACSL Example Problem (**Tangerine Code**) (https://youtu.be/5P5iK-5heEc)

The video walks through the solution to a 2-variable recursive function, that is, $f(x, y) = \{ \ldots$.

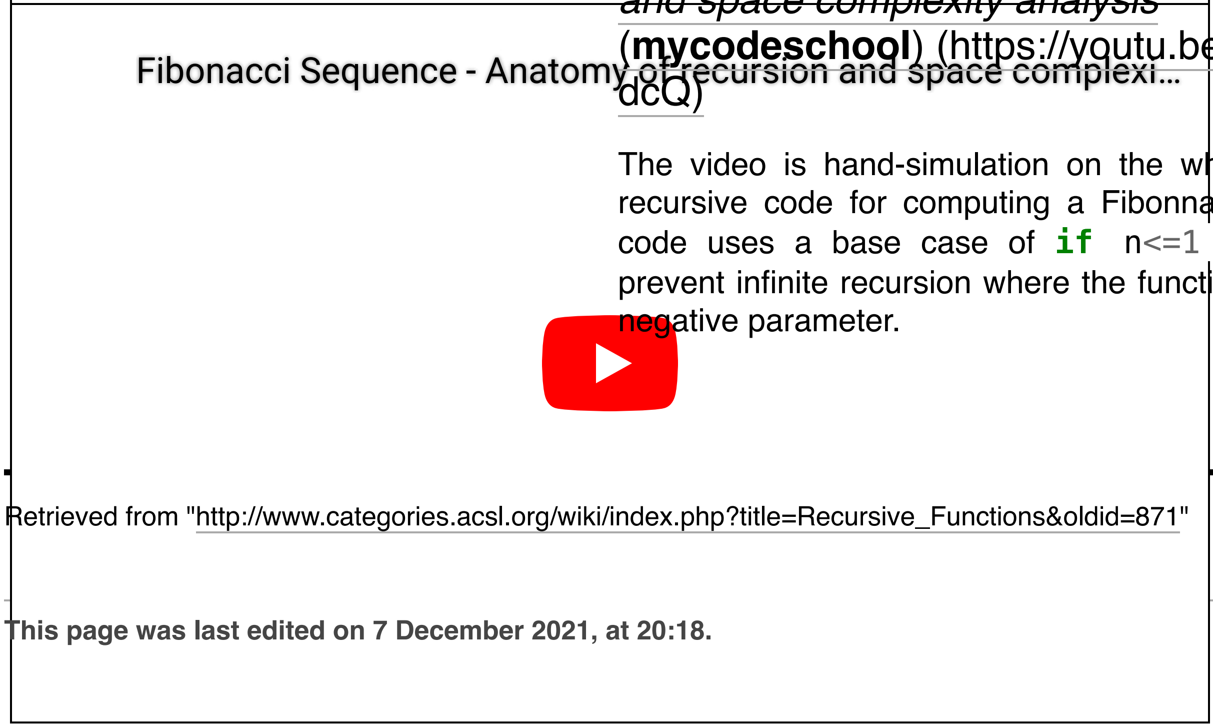
Other Videos

Because recursion is such a fundamental concept in computer science, there is no end to the number of YouTube videos that cover the topic. There are dozens of walk-throughs of the factorial and Fibonacci functions, and of other classic algorithms such as Towers of Hanoi and Binary Trees. Some of the videos contain ads; ACSL is not responsible for the ads and does not receive compensation in any form for those ads.



Recursion lecture 1 (**anim aland**) (https://youtu.be/LdNdfPhwCP4)

Prof. Alan Dorin from Monash University presents an wonderful introduction to recursion. The last part of the video uses factorial as an example. Taken note how his base case guards against infinite recursion.


Fibonacci Sequence - Anatomy of recursion and space complexi...

Fibonacci Sequence - Anatomy of recursion and space complexity analysis (**mycodeschool**) (https://youtu.be/dxyYP3BSdcQ)

The video is hand-simulation on the whiteboard of the recursive code for computing a Fibonnaci number. The code uses a base case of `if n<=1 return n` to prevent infinite recursion where the function called with a negative parameter.

Retrieved from "http://www.categories.acsl.org/wiki/index.php?title=Recursive_Functions&oldid=871"

This page was last edited on 7 December 2021, at 20:18.