

Extracted from:

A Common-Sense Guide to Data Structures and Algorithms

Level Up Your Core Programming Skills

This PDF file contains pages extracted from *A Common-Sense Guide to Data Structures and Algorithms*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

A Common-Sense Guide to Data Structures and Algorithms

Level Up Your Core
Programming Skills



Jay Wengrow
edited by Brian MacDonald

A Common-Sense Guide to Data Structures and Algorithms

Level Up Your Core Programming Skills

Jay Wengrow

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at <https://pragprog.com>.

The team that produced this book includes:

Publisher: Andy Hunt

VP of Operations: Janet Furlow

Executive Editor: Susannah Davidson Pfalzer

Development Editor: Brian MacDonald

Copy Editor: Nicole Abramowitz

Indexing: Potomac Indexing, LLC

Layout: Gilson Graphics

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-244-2

Book version: P2.0—July 2018

Recursion in Action

While previous examples of the NASA countdown and calculating factorials can be solved with recursion, they can also be easily solved with classical loops. While recursion is interesting, it doesn't really provide an advantage when solving these problems.

However, recursion is a natural fit in any situation where you find yourself having to repeat an algorithm within the same algorithm. In these cases, recursion can make for more readable code, as you're about to see.

Take the example of traversing through a filesystem. Let's say that you have a script that does something with every file inside of a directory. However, you don't want the script to only deal with the files inside the *one* directory—you want it to act on all the files within the *subdirectories* of the directory, and the subdirectories of the subdirectories, and so on.

Let's create a simple Ruby script that prints out the names of all subdirectories within a given directory.

```
def find_directories(directory)
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
      filename != "." && filename != ".."
      puts "#{directory}/#{filename}"
    end
  end
end

# Call the find_directories method on the current directory:
find_directories(".")
```

In this script, we look through each file within the given directory. If the file is itself a subdirectory (and isn't a single or double period, which represent the current and previous directories, respectively), we print the subdirectory name.

While this works well, it only prints the names of the subdirectories *immediately* within the current directory. It does not print the names of the subdirectories *within* those subdirectories.

Let's update our script so that it can search one level deeper:

```
def find_directories(directory)
  # Loop through outer directory:
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
      filename != "." && filename != ".."
      puts "#{directory}/#{filename}"
    end
  end
end
```

```

# Loop through inner subdirectory:
Dir.foreach("#{directory}/#{filename}") do |inner_filename|
  if File.directory?("#{directory}/#{filename}/#{inner_filename}") &&
    inner_filename != "." && inner_filename != ".."
    puts "#{directory}/#{filename}/#{inner_filename}"
  end
end
end
end
end

# Call the find_directories method on the current directory:
find_directories(".")

```

Now, every time our script discovers a directory, it then conducts an identical loop through the subdirectories of *that* directory and prints out the names of the subdirectories. But this script also has its limitations, because it's only searching two levels deep. What if we wanted to search three, four, or five levels deep? What if we wanted to search as deep as our subdirectories go? That would seem to be impossible.

And *this* is the beauty of recursion. With recursion, we can write a script that goes arbitrarily deep—and is also simple!

```

def find_directories(directory)
  Dir.foreach(directory) do |filename|
    if File.directory?("#{directory}/#{filename}") &&
      filename != "." && filename != ".."
      puts "#{directory}/#{filename}"
      find_directories("#{directory}/#{filename}")
    end
  end
end

# Call the find_directories method on the current directory:
find_directories(".")

```

As this script encounters files that are themselves subdirectories, it calls the `find_directories` method upon that very subdirectory. The script can therefore dig as deep as it needs to, leaving no subdirectory unturned.

To visually see how this algorithm applies to an example filesystem, examine the [diagram on page 7](#), which specifies the order in which the script traverses the subdirectories.

Note that recursion in a vacuum does not necessarily speed up an algorithm's efficiency in terms of Big O. However, we will see in the following chapter that recursion can be a core component of algorithms that *does* affect their speed.

