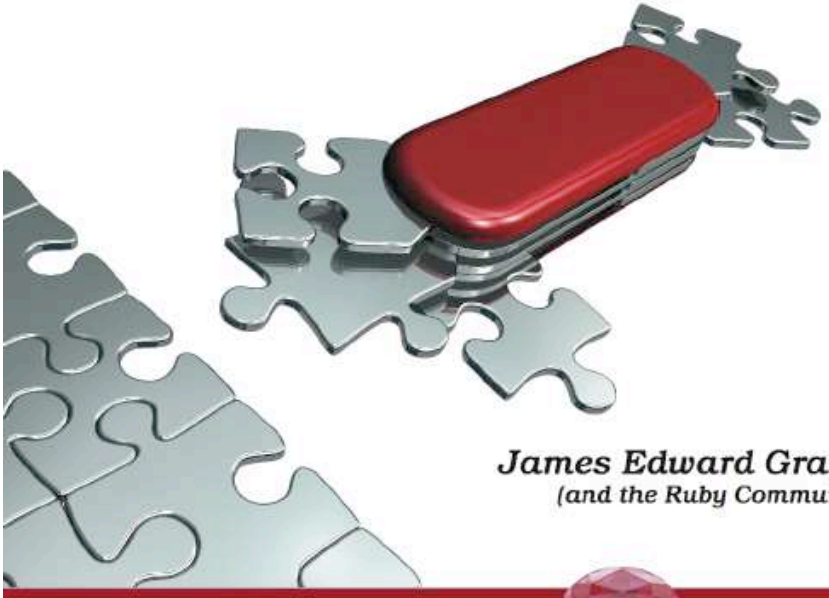
Extracted from:

Best of Ruby Quiz

Volume One

The Pragmatic Programmers

Best of Ruby Quiz



James Edward Gray II
(and the Ruby Community)

The Facets of Ruby Series

Answer 13
From page 31

1-800-THE-QUIZ

Some problems are just easier to express with recursion. For me, this is one of those problems.

If you're not familiar with the idea, *recursion* is defining a method that calls itself. Sometimes we humans struggle to understand this concept of defining something in terms of itself, but it can make some programming challenges easier. Let's use this problem to explore the possibilities of recursion.

Word Signatures

The first step to solving this problem is doing the right work when you read in the dictionary. Come search time, we won't be interested in words at all, just groupings of digits. Each word in the dictionary can be encoded as the digits we would need to type on a phone. If we do that while we're reading them in and store them correctly, we can save ourselves much work down the road. First, let's begin a PhoneDictionary object and give it an encoding:

1_800_the_quiz/phone_words.rb
```
require "enumerator"
class PhoneDictionary
  def self.encode( letter )
    case letter.downcase
    when "a", "b", "c"       then "2"
    when "d", "e", "f"       then "3"
    when "g", "h", "i"       then "4"
    when "j", "k", "l"       then "5"
    when "m", "n", "o"       then "6"
    when "p", "q", "r", "s" then "7"
    when "t", "u", "v"       then "8"
    when "w", "x", "y", "z" then "9"
    end
  end
end
```

Beware of Recursion

Though it simplifies some problems, recursion has its price. First, the repeated method calls can be slow. Depending on the size of the data you are crunching, you may feel the slowdown. Run the code in this chapter against different-sized dictionaries, and you'll start to see the penalty.

Ruby also uses the C stack, which may not be set very deep by default, so it's best to avoid problems that need a lot of nested calls. The examples in this chapter are fine, because they never go deeper than eight levels. Make sure you stay aware of the limits in your own code.

There's no such thing as recursive code that can't be unrolled to work as an iterative solution. If the restrictions bite you, you may just have to do the extra work.

My first instinct was to put the encoding into a constant, but I later decided a method would make it easy to replace (without a warning from Ruby). Not all phones are like mine, after all.

Obviously, you just give this method a letter, and it will give you back the digit for that letter.

Now, we need to set up our dictionary data structure. As with the rest of the methods in this quiz, this is an instance method in our PhoneDictionary class.

1_800_the_quiz/phone_words.rb
```
def initialize( word_file )
  @words = Hash.new { |dict, digits| dict[digits] = Array.new }
  ("0".."9").each { |n| @words[n] << n }
  %w{a i}.each { |word| @words[self.class.encode(word)] << word }

  warn "Loading dictionary..." if $DEBUG
  read_dictionary(word_file)
end
```

I use a Hash to hold word groups. A group is identified by the digit encoding (hash key) and is an Array of all words matching that encoding (hash value). I use Hash's default block parameter to create word group arrays as needed.

The next line is a trick to ease the searching process. Since it's possible

for numbers to be left in, I decided to just turn individual numbers into words. This will allow bogus solutions with many consecutive numbers, but those are easily filtered out after the search.

Finally, I plan to filter out individual letter words, which many dictionaries include. Given that, I add the only single-letter words that make sense to me, careful to use encoding() to convert them correctly.[35]

At the bottom of that method, you can see the handoff to the dictionary parser:[36]

1_800_the_quiz/phone_words.rb
```
def read_dictionary( dictionary )
  File.foreach(dictionary) do |word|
    word.downcase!
    word.delete!("^a-z")

    next if word.empty? or word.size < 2 or word.size > 7

    chars  = word.enum_for(:each_byte)
    digits = chars.map { |c| self.class.encode(c.chr) }.join

    @words[digits] << word unless @words[digits].include?(word)
  end
end
```

This method is just a line-by-line read of the dictionary. I normalize the words to a common case[37] and toss out punctuation and whitespace. The method skips any words below two characters in length as well as any more than seven. Finally, words are split into characters, using the handy enum_for() from the Enumerator library (see the sidebar, on page 157, for details), and then digit encoded and added to the correct group. The code first verifies that a word wasn't already in the group, though, ensuring that our transformations don't double up any words.

The Search

With setup out of the way, we are ready to search a given phone number for word matches. First, we need a simple helper method that checks

[35] Be warned, this step assumes we are dealing with an American English dictionary.

[36] Notice the $DEBUG message hidden in this section of code. Ruby will automatically set that variable to true when passed the -d command-line switch, so it's a handy way to embed trace instructions you may want to see during debugging.

[37] Even though we're going to end up with uppercase results, I generally normalize case down, not up. Some languages make distinctions between concepts like title case and uppercase, so downcasing is more consistent.

a digit sequence against the beginning of a number. If it matches, we want it to return what's left of the original number:

1_800_the_quiz/phone_words.rb
```
def self.match( number, digits )
  if number[0, digits.length] == digits
    number[digits.length..-1]
  else
    nil
  end
end
```

With that, we are finally ready to search:

1_800_the_quiz/phone_words.rb
```
def search( number, chunks = Array.new )
  @words.inject(Array.new) do |all, (digits, words)|
    if remainder = self.class.match(number, digits)
      new_chunks = (chunks.dup << words)
      if remainder.empty?
        all.push(new_chunks)
      else
        all.push(*search(remainder, new_chunks))
      end
    else
      all
    end
  end
end
```

The idea here is to match numbers against the front of the phone number, passing the matched words and what's left of the String down recursively, until there is nothing left to match.

The method returns an Array of chunks, each of which is an Array of all the words that can be used at that point. For example, a small part of the search results for the quiz example shows that the number could start with the word *USER* followed by *-8-AX*, *TAX*, or other options:

```
[...
  [["user"], ["8"], ["aw", "ax", "ay", "by"]],
  [["user"], ["taw", "tax", "tay"]],
...]
```

The recursion keeps this method short and sweet, though you may need to work through the flow a few times to understand it.

The key to successful recursion is always having an *exit condition*, the point at which you stop recursing. Here, the method recurses only when there are remaining digits in the number. Once we've matched them all or failed to find any matches, we're done.

Enumerator: A Hidden Treasure

The Enumerator library is a hidden treasure of Ruby's standard library that was undocumented until very recently. Here's a quick tour to get you started using it today.

The main function of the library is to add an enum_for() method to Object, also aliased as to_enum(). Call this method, passing a method name and optionally some parameters, and you'll receive an Enumerable object using the passed method as each(). As you can see in the dictionary-parsing code of this chapter, that's a handy tool for switching Strings to iterate over characters, among other uses.

As an added bonus, the library adds two more iterators to Enumerable:

```
>> require "enumerator"
=> true
>> (1..10).each_slice(2) { |slice| p slice }
[1, 2]
[3, 4]
[5, 6]
[7, 8]
[9, 10]
=> nil
>> (1..10).each_cons(3) { |consecutive| p consecutive }
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
[6, 7, 8]
[7, 8, 9]
[8, 9, 10]
=> nil
```

Cleaning Up and Showing Results

Obviously the results returned from the search aren't printable as they stand. Let's use some more recursion to flatten the nested arrays down to strings.

1_800_the_quiz/phone_words.rb

```
def chunks_to_strings( chunks )
  chunk, *new_chunks = chunks.dup
  if new_chunks.empty?
    chunk.map { |word| word.upcase }
  else
    chunk.map do |word|
```

```
      chunks_to_strings(new_chunks).map { |words| "#{word.upcase}-#{words}" }
    end.flatten
  end
end
```

Again the idea behind this method is trivial: peel a single word group off, and combine it with all the other combinations generated through recursion of the remaining groups. Logically, the exit condition here is when we reach the final word group, and we can just return those words when that happens.

The class requires just one more public interface method to tie it all together:

1_800_the_quiz/phone_words.rb
```
def number_to_words( phone_number )
  warn "Searching..." if $DEBUG
  results = search(phone_number)

  warn "Preparing output..." if $DEBUG
  results.map! { |chunks| chunks_to_strings(chunks) }
  results.flatten!
  results.reject! { |words| words =~ /\d-\d/ }
  results.sort!

  results
end
```

This method runs the workflow. Perform a search, convert the results to Strings, remove bogus results, clean up, and return the fruits of our labor. A caller of this method provides a phone number and receives ready-to-print word replacements.

Here's the last bit of code that implements the quiz interface:

1_800_the_quiz/phone_words.rb
```
if __FILE__ == $0
  dictionary = if ARGV.first == "-d"
    ARGV.shift
    PhoneDictionary.new(ARGV.shift)
  else
    PhoneDictionary.new("/usr/share/dict/words")
  end

  ARGF.each_line do |phone_number|
    puts dictionary.number_to_words(phone_number.delete("^0-9"))
  end
end
```

Additional Exercises

1. Unroll the search() method presented in this chapter to build an iterative solution.
2. Benchmark the recursion and iterative versions of the code. What was the speed increase?

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

Best of Ruby Quiz
`pragmaticprogrammer.com/titles/fr_quiz`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/fr_quiz.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com