Ruby Quiz

Word Chains (#44)

Our own Dave Thomas has also posted some Ruby code challenges on his blog in the past. There are several interesting problems there:

CodeKata

This week's Ruby Quiz is one of my favorite problems posted by Dave.

Given two words of equal length as command-line arguments, build a chain of words connecting the first to the second. Each word in the chain must be in the dictionary and every step along the chain changes exactly one letter from the previous word.

Again, your program should accept input as two command-line arguments. Your program should also allow a -d command-line switch followed by a dictionary file to use, but feel free to choose a sensible default for your system. The result should be a word chain starting with the first word and ending with the last printed to STDOUT, one word per line. Print an error message if a chain cannot be found.

Bonus points for finding the shortest chain and/or a small execution time.

Here's an example:

```
$ time ruby -d word_chain.rb duck ruby
Loading dictionary...
Building chain...
duck
ruck
rusk
ruse
rube
ruby

real    0m27.551s
user    0m27.402s
sys     0m0.113s
```

Quiz Summary

Gavin Kistner asked that I try timing the quiz solutions this week. I did try, but ran into some difficulties I didn't have time to short out. Below are the results I got for some of the submissions. (Simon Kröger later worked up a complete set of time trials.)

```
=== Timing ./Brian Schroeder/wordchain.rb ===
Loading database
Searching connection between duck and ruby
duck
dusk
rusk
ruse
rube
ruby

=== ./Brian Schroeder/wordchain.rb:  3.722559 seconds ===

=== Timing ./Dominik Bathon/word_chain.rb ===
duck
luck
luce
lube
rube
ruby
```

```
=== ./Dominik Bathon/word_chain.rb:  1.472335 seconds ===

=== Timing ./James Edward Gray II/word_chain.rb ===
duck
ruck
rusk
ruse
rube
ruby
=== ./James Edward Gray II/word_chain.rb:  27.577392 seconds ===

=== Timing ./Levin Alexander/word_chains.rb ===
duck
ruck
rusk
ruse
rube
ruby
=== ./Levin Alexander/word_chains.rb:  2.795315 seconds ===

=== Timing ./Will Thimbleby/word_chain.rb ===
duck
dunk
dune
rune
rube
ruby
=== ./Will Thimbleby/word_chain.rb:  63.026336 seconds ===
```

I actually expected Brian's to be the fastest, just from what I had read about them as they came in. Under the hood, Brian is using a priority queue written in C. As the saying goes though, the speed is in the algorithm, and Dominik and Levin prove the truth of it.

For an interesting comparison, Levin is using a plain Ruby priority queue. Let's take a look at that class:

```ruby
# inefficient implementation of a priority queue
#
class SimpleQueue
  def initialize
    @storage = Hash.new { [] }
  end
  def insert(priority, data)
    @storage[priority] = @storage[priority] << data
  end
  def extract_min
    return nil if @storage.empty?
    key, val = *@storage.min
    result = val.shift
    @storage.delete(key) if val.empty?
    return result
  end
end
```

If you look at that insert() method, you might find the calls a bit odd. The code does work, but only because of the awkward assignment that shouldn't be needed. This is a gotcha that bit me early in learning Ruby so I'll explain it here in the hope of helping others.

Array.new() can take a number and a block. It will invoke the block the indicated number of times to generate an Array, using the return value of the block as each member.

Because Hash.new() also takes a block and will call it when a key is first accessed without an assignment, the natural assumption is that it uses the return value, and in truth it does, but it does not set the key to that value! That's why the extra assignment is needed above.

The fix is to use the passed in Hash and key String objects to set it yourself. Using that, we can write the above a little more naturally:

```ruby
```

```
    # inefficient implementation of a priority queue
    #
    class SimpleQueue
      def initialize
        @storage = Hash.new { |hash, key| hash[key] = [] }
      end
      def insert(priority, data)
        @storage[priority] << data
      end
      def extract_min
        return nil if @storage.empty?
        key, val = *@storage.min
        result = val.shift
        @storage.delete(key) if val.empty?
        return result
      end
    end
```

I just think that's more natural and easy to follow. They work the same.

Looking at the code itself, there's nothing too fancy here. It stores items by priority in a Hash. The real work is done in extract_min() which just locates the minimum value, shifts some data off of that Array, and returns it. The comment warns that it's inefficient, but it sure is easy to setup and use. Hard to beat that for just fifteen lines of code. Nice work Levin.

Now I want to examine Dominik's lightning fast solution. Here's how it starts:

```ruby
    DEFAULT_DICTIONARY = "/usr/share/dict/words"

    # Data structure that efficiently stores words from a dictionary in a
way,
    # that it is easy to find all words that differ from a given word only
at
    # one letter (words that could be the next step in a word chain).
    # Example: when adding the word "dog", add_word will register "dog" as
    # step for "\0og", "d\0g" and "do\0", later each_possible_step("cat")
    # will yield all words registered for "\0at", "c\0t" or "ca\0".
    class WordSteps
        def initialize
            @steps = Hash.new { |h, k| h[k] = [] }
            @words = {}
        end

        # yields all words (as strings) that were added with add_word
        def each_word(&block)
            @words.each_key(&block)
        end

        # add all steps for word (a string) to the steps
        def add_word(word)
            sym = word.to_sym
            wdup = word.dup
            for i in 0...word.length
                wdup[i] = 0
                @steps[wdup] << sym
                wdup[i] = word[i]
            end
            @words[word] = sym # for allow_shorter and each_word
        end

        # yields each possible next step for word (a string) as symbol,
some
        # possible steps might be yielded multiple times
        # if allow_shorter is true, word[0..-2].to_sym will also be yielded
        # if available
        # if allow_longer is true, all words that match /#{word}./ will be
        # yielded
        def each_possible_step(word, allow_shorter = false,
                                     allow_longer = false)
            wdup = word.dup
            for i in 0...word.length
```

```
                wdup[i] = 0
                if @steps.has_key?(wdup)
                    @steps[wdup].each { |step| yield step }
                end
                wdup[i] = word[i]
            end
            if allow_shorter && @words.has_key?(tmp = word[0..-2])
                yield @words[tmp]
            end
            if allow_longer && @steps.has_key?(tmp = word + "\0")
                @steps[tmp].each { |step| yield step }
            end
        end

        # ...
```

The comments are just great in this code. If you read them, you'll understand how the code moves so darn fast. Here's the mini-summary: When called add_word() maps a word to all possible variations with exactly one letter changed to a null character. Later, each_possible_step() can use the same mapping to quickly look up all possibilities for the current word in question.

This can also handle searches where words aren't the same size, though that wasn't part of the quiz.

```ruby
        # ...

        # tries to find a word chain between word1 and word2 (strings)
using
        # all available steps
        # returns the chain as array of symbols or nil, if no chain is
found
        # shorter/longer determines if shorter or longer words are allowed
in
        # the chain
        def build_word_chain(word1, word2, shorter = false, longer = false)
            # build chain with simple breadth first search
            current = [word1.to_sym]
            pre = { current[0] => nil } # will contain the predecessors
            target = word2.to_sym
            catch(:done) do
                until current.empty?
                    next_step = []
                    current.each do |csym|
                        each_possible_step(csym.to_s, shorter, longer) do
|ssym|
                            # have we seen this word before?
                            unless pre.has_key? ssym
                                pre[ssym] = csym
                                throw(:done) if ssym == target
                                next_step << ssym
                            end
                        end
                    end
                    current = next_step
                end
                return nil # no chain found
            end
            # build the chain (in reverse order)
            chain = [target]
            chain << target while target = pre[target]
            chain.reverse
        end

        # ...
```

This is the search for a chain. Believe it or not, it's a rather boring unidirectional breadth-first search. Most people implemented much fancier searches, but thanks to Domink's dictionary storage this code doesn't need to be clever.

This code just uses each_possible_step() to walk level-by-level of like words, until it finds the end word. The pre Hash is used to keep the code from retracing its steps and to walk the previous word chain to build the final answer at the bottom of the method.

This method has a nice use of catch() and throw() to create the equivalent of a labeled goto call in many other languages.

There's one more piece to this class:

```ruby
        # ...

        # builds and returns a WordSteps instance "containing" all words
with
        # length in length_range from the file file_name
        def self.load_from_file(file_name, length_range)
            word_steps = new
            IO.foreach(file_name) do |line|
                # only load words with correct length
                if length_range === (word = line.strip).length
                    word_steps.add_word(word.downcase)
                end
            end
            word_steps
        end
    end

    # ...
```

Here's the simple dictionary reading method. Note the clever use of a Range argument here, to support word chains of differing sizes. The === check ensures that the current dictionary word is in the Range we care about, before it's added to the memory mappings.

Finally, here's the interface code:

```ruby
    # ...

    if $0 == __FILE__
        dictionary = DEFAULT_DICTIONARY

        # parse arguments
        if ARGV[0] == "-d"
            ARGV.shift
            dictionary = ARGV.shift
        end
        unless ARGV.size == 2
            puts "usage: #$0 [-d path/to/dictionary] word1 word2"
            exit 1
        end
        word1, word2 = ARGV[0].strip.downcase, ARGV[1].strip.downcase

        shorter = word1.length > word2.length
        longer = word1.length < word2.length
        length_range = if longer
            word1.length..word2.length
        else
            word2.length..word1.length
        end

        # read dictionary
        warn "Loading dictionary..." if $DEBUG
        word_steps = WordSteps.load_from_file(dictionary, length_range)
        word_steps.add_word(word2) # if it is not in dictionary

        # build chain
        warn "Building chain..." if $DEBUG
        chain = word_steps.build_word_chain(word1, word2, shorter, longer)

        # print result
        puts chain || "No chain found!"
```

```
end
```

Most of that is just the code to support the arguments from the quiz. Note the clever building of the length Range that allows the program to switch behavior when different sized words are given. All around great code Domink. Thanks for the lesson!

My thanks to all who were able to understand my command-line argument instructions this week. ;) Seriously, thanks to all submitters. Many great solutions this week.

If you don't know what tomorrow's quiz is yet, you're not reading Redhanded closely enough...