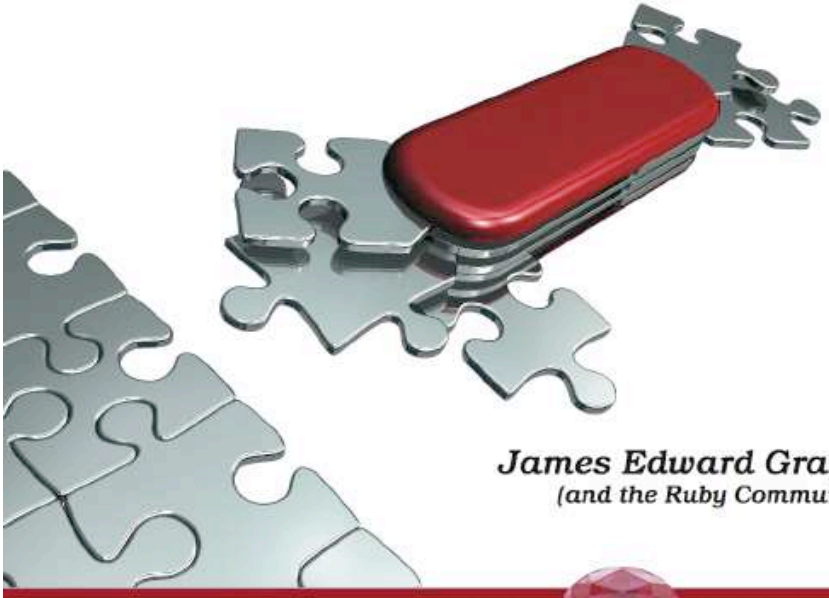
Extracted from:

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 23
From page 53

Countdown

At first glance, the search space for this problem looks very large. The six source numbers can be ordered various ways, and you don't have to use all the numbers. Beyond that, you can have one of four operators between each pair of numbers. Finally, consider that `1 * 2 + 3` is different from `1 * (2 + 3)`. That's a lot of combinations.

However, we can prune that large search space significantly. Let's start with some simple examples and work our way up. Addition and multiplication are commutative, so we have this:

```
1 + 2 = 3 and 2 + 1 = 3
1 * 2 = 2 and 2 * 1 = 2
```

We don't need to handle it both ways. One will do.

Moving on to numbers, the example in the quiz used two 5s as source numbers. Obviously, these two numbers are interchangeable. The first 5 plus 2 is 7, just as the second 5 plus 2 is 7.

What about the possible source number 1? Anything times 1 is itself, so there is no need to check multiplication of 1. Similarly, anything divided by 1 is itself. No need to divide by 1.

Let's look at 0. Adding and subtracting 0 is pointless. Multiplying by 0 takes us back to 0, which is pretty far from a number from 100 to 999 (our goal). Dividing 0 by anything is the same story, and dividing by 0 is illegal, of course. Conclusion: 0 is useless. Now, you can't get 0 as a source number; but, you can safely ignore any operation(s) that result in 0.

Those are all single-number examples, of course. Time to think bigger. What about negative numbers? Our goal is somewhere from 100 to

999. Negative numbers are going the wrong way. They don't help, so you can safely ignore any operation that results in a negative number.

Finally, consider this:

```
(5 + 5) / 2 = 5
```

The previous is just busywork. We already had a 5; we didn't need to make one. Any operations that result in one of their operands can be ignored.

Using simplifications like the previous, you can get the search space down to something that can be brute-force searched pretty quickly, as long as we're dealing only with six numbers.

Pruning Code

Dennis Ranke submitted the most complete example of pruning, so let's start with that. Here's the code:

countdown/pruning.rb

```
class Solver
  class Term
    attr_reader :value, :mask

    def initialize(value, mask, op = nil, left = nil, right = nil)
      @value = value
      @mask = mask
      @op = op
      @left = left
      @right = right
    end

    def to_s
      return @value.to_s unless @op
      "(#@left #@op #@right)"
    end
  end

  def initialize(sources, target)
    printf "%s -> %d\n", sources.inspect, target
    @target = target
    @new_terms = []
    @num_sources = sources.size
    @num_hashes = 1 << @num_sources

    # the hashes are used to check for duplicate terms
    # (terms that have the same value and use the same
    # source numbers)
    @term_hashes = Array.new(@num_hashes) { {} }
```

```
    # enter the source numbers as (simple) terms
    sources.each_with_index do |value, index|

      # each source number is represented by one bit in the bit mask
      mask = 1 << index
      p mask
      p value
      term = Term.new(value, mask)
      @new_terms << term
      @term_hashes[mask][value] = term
    end
  end
end
```

The Term class is easy enough. It is used to build tree-like representations of math operations. A Term can be a single number or @left Term, @right Term, and the @op joining them. The @value of such a Term would be the result of performing that math.

The tricky part in this solution is that it uses bit masks to compare Terms. The mask is just a collection of bit switches used to represent the source numbers. The bits correspond to the index for that source number. You can see this being set up right at the bottom of initialize().

These mask-to-Term pairs get stored in @term_hashes. This variable holds an Array, which will be indexed with the mask of source numbers in a Term. For example, an index mask of 0b000101 (5 in decimal) means that the first and third source numbers are used, which are index 0 and 2 in both the binary mask and the source list.

Inside the Array, each index holds a Hash. Those Hashes hold decimal value to Term pairs. The values are numbers calculated by combining Terms. For example, if our first source number is 100 and the second is 2, the Hash at Array index 0b000011 (3) will eventually hold the keys 50, 98, 102, and 200. The values for these will be the Term objects showing the operators needed to produce the number.

All of this bit twiddling is very memory efficient. It takes a lot less computer memory to store 0b000011 than it does [100, 2].

countdown/pruning.rb

```
class Solver
  def run
    collision = 0
    best_difference = 1.0/0.0
    next_new_terms = [nil]
    until next_new_terms.empty?
      next_new_terms = []
```

```
# temporary hashes for terms found in this iteration
# (again to check for duplicates)
new_hashes = Array.new(@num_hashes) { {} }

# iterate through all the new terms (those that weren't yet used
# to generate composite terms)
@new_terms.each do |term|

  # iterate through the hashes and find those containing terms
  # that share no source numbers with 'term'
  index = 1
  term_mask = term.mask

  # skip over indices that clash with term_mask
  index += collision - ((collision - 1) & index) while
              (collision = term_mask & index) != 0
  while index < @num_hashes
    hash = @term_hashes[index]

    # iterate through the hashes and build composite terms using
    # the four basic operators
    hash.each_value do |other|
      new_mask = term_mask | other.mask
      hash = @term_hashes[new_mask]
      new_hash = new_hashes[new_mask]

      # sort the source terms so that the term with the larger
      # value is left
      # (we don't allow fractions and negative subterms are not
      # necessairy as long as the target is positive)
      if term.value > other.value
        left_term = term
        right_term = other
      else
        left_term = other
        right_term = term
      end
      [:+, :-, :*, :/].each do |op|

        # don't allow fractions
        next if op == :/ &&
                left_term.value % right_term.value != 0

        # calculate value of composite term
        value = left_term.value.send(op, right_term.value)

        # don't allow zero
        next if value == 0

        # ignore this composite term if this value was already
```

```
            # found for a different term using the same source
            # numbers
            next if hash.has_key?(value) || new_hash.has_key?(value)

            new_term = Term.new(value, new_mask, op, left_term,
                                right_term)

            # if the new term is closer to the target than the
            # best match so far print it out
            if (value - @target).abs < best_difference
              best_difference = (value - @target).abs
              printf "%s = %d (error: %d)\n", new_term, value,
                                 best_difference
              return if best_difference == 0
            end

            # remember the new term for use in the next iteration
            next_new_terms << new_term
            new_hash[value] = new_term
          end
        end
        index += 1
        index += collision - ((collision - 1) & index) while
                 (collision = term_mask & index) != 0
      end
    end

    # merge the hashes with the new terms into the main hashes
    @term_hashes.each_with_index do |hash, index|
      hash.merge!(new_hashes[index])
    end

    # the newly found terms will be used in the next iteration
    @new_terms = next_new_terms
  end
 end
end
```

That's very well-commented code, so I won't bother to break it all down. I do want to point out a few things, though.

This method repeatedly walks through all of the @new_terms, combining them with all the already found @term_hashes to reach new values. At each step we build up a collection of next_new_terms that will replace @new_terms when the process loops. Also being loaded is new_hashes, which will be merged into @term_hashes, giving us more to expand on in the next iteration.

Be sure to spot the two pieces of code for avoiding collisions. If we find ourselves working with an index that matches the term_mask at any

point, we know we are duplicating work because we are working with the same source list. In these cases, index gets bumped to move us along.

The rest of the method is the pruning work we looked into at the start of this discussion. The comments will point out what each section of code is skipping.

Here's the code you need to turn all that work into a solution:

countdown/pruning.rb
```
if ARGV[0] && ARGV[0].downcase == 'random'
  ARGV[0] = rand(900) + 100
  ARGV[1] = (rand(4) + 1) * 25
  5.times {|i| ARGV[i + 2] = rand(10) + 1}
end

if ARGV.size < 3
  puts "Usage: ruby #$0 <target> <source1> <source2> ..."
  puts "   or: ruby #$0 random"
  exit
end

start_time = Time.now
Solver.new(ARGV[1..-1].map {|v| v.to_i}, ARGV[0].to_i).run
printf "%f seconds\n", Time.now - start_time
```

The previous solution is lightning fast. Run it a few times to see for yourself. It can work so fast because heavy pruning allows it to skip a lot of useless operations.

Coding Different Strategies

Next, I want to look at Brian Schröder's solution. I won't show the whole thing here because it's quite a lot of code. However, it can switch solving methods as directed and even solve using fractions. Here's the heart of it:

countdown/countdown.rb
```
# Search all possible terms for the ones that fit best.

# Systematically create all terms over all subsets of the set of numbers in
# source, and find the one that is closest to target.
#
# Returns the solution that is closest to the target.
#
# If a block is given, calls the block each time a better or equal solution
# is found.
#
# As a heuristic to guide the search, sort the numbers ascending.
```

```
def solve_countdown(target, source, use_module)
  source = source.sort_by{|i|-i}
  best = nil
  best_distance = 1.0/0.0
  use_module::each_term_over(source) do | term |
    distance = (term.value - target).abs
    if distance <= best_distance
      best_distance = distance
      best = term
      yield best if block_given?
    end
  end
  return best
end
```

This method takes the target and source numbers in addition to a Module (which I'll return to in a minute) as parameters. The first line is the sort mentioned in the comment. Then best and best_distance are initialized to nil and infinity (1.0/0.0) to track the best solution discovered so far.

After the setup, the method calls into the each_term_over() method, provided by the Module it was called with. The Module to use is determined by the interface code (not shown) based on the provided command-line switches. There are four possible choices. Two deal with fractions while two are integer only, and there is a recursive and "memoized" version for each number type. The program switches solving strategies based on the user's requests. (This is a nice use of the Strategy design pattern.)

Here is each_term_over() in the ModuleRecursive::Integral:

countdown/countdown-recursive.rb

```
module Recursive
  # Allow only integral results
  module Integral
    # Call the given block for each term that can be constructed over a set
    # of numbers.
    #
    # Recursive implementation that calls a block each time a new term has been
    # stitched together. Returns each term multiple times.
    #
    # This version checks that only integral results may result.
    #
    # Here I explicitly coded the operators, because there is not
    # much redundance.
    #
    # This may be a bit slow, because it zips up through the whole callstack
    # each time a new term is created.
```

```
    def Integral.each_term_over(source)
      if source.length == 1
        yield source[0]
      else
        source.each_partition do | p1, p2 |
          each_term_over(p1) do | op1 |
            yield op1
            each_term_over(p2) do | op2 |
              yield op2
              if op2.value != 0
                yield Term.new(op1, op2, :+)
                yield Term.new(op1, op2, :-)
                yield Term.new(op1, op2, :'/') if op2.value != 1 and
                                                  op1.value % op2.value == 0
              end
              if op1.value != 0
                yield Term.new(op2, op1, :-)
                if op1.value != 1
                  yield Term.new(op2, op1, :'/') if op2.value % op1.value == 0
                  yield Term.new(op1, op2, :*) if op2.value != 0 and
                                                  op2.value != 1
                end
              end
            end
          end
        end
      end
    end
  end
end
```

This method recursively generates terms in every possible combination. This is a key point to a working solution. If you try adding a number at a time, you generate solutions looking like these:

```
(((num op num) op num) op num)...
```

A tricky example posted to Ruby Talk by daz, "Target: 926, Source: 75, 2, 8, 5, 10, 10," shows off the folly of this approach. The only answer is the following:

```
(75 - 5 + 8) * (2 + 10) - 10
```

As you can see, the 2 + 10 term must be built separately from the 75 - 5 + 8 term, and then the two can be combined.

Getting back to the previous code, the each_partition() method it uses was added to Array in a different section of the code (not shown). It returns “each true partition (containing no empty set) exactly once.”

Term objects (not shown) just manage their operands and operator, providing mainly String representation and result evaluation.

The block we're yielding to is the block passed by solve_countdown(), which we examined earlier. It simply keeps track of the best solution generated so far.

The interesting part of all this is the same method in a different module. The listing on the next page is the each_term_over() method from Memoized::Integral.

The result of this method is the same, but it uses a technique called *memoization* to work faster. When Terms are generated in here, they get added to the Hash memo. After that, all the magic is in the very first line, which simply skips all the work the next time those source numbers are examined.

This trades memory (the Hash of stored results) for speed (no repeat work). That's why the solution provides other options too. Maybe the target platform won't have the memory to spare. This is a handy technique showcased in a nice implementation.

Additional Exercises

1. Try adding some pruning or memoization to your solution. Time solving the same problem before and afterward to see if whether speeds up the search.
2. You can find a great web-based interactive solver for this number game at http://www.crosswordtools.com/numbers-game/. Extend your solution to provide a similar web interface.

countdown/countdown-memoized.rb

```
module Memoized
  module Integral
    # Call the given block for each term that can be constructed over
    # a set of numbers.
    #
    # Recursive implementation that calls a block each time a new term
    # has been stitched together.  Returns each term multiple times.
    #
    # This version checks that only integral results may result.
    #
    # Here I explicitly coded the operators, because there is not much
    # redundance.
    #
    # This may be a bit slow, because it zips up through the whole
    # callstack each time a new term is created.
    def Integral.each_term_over(source, memo = {}, &block)
      return memo[source] if memo[source]

      result = []
      if source.length == 1
        result << source[0]
      else
        source.each_partition do | p1, p2 |
          each_term_over(p1, memo, &block).each do | op1 |
            each_term_over(p2, memo, &block).each do | op2 |
              if op2.value != 0
                result << Term.new(op1, op2, :+)
                result << Term.new(op1, op2, :-)
                result << Term.new(op1, op2, :'/') if op2.value != 1 and
                                                      op1.value % op2.value == 0
              end
              if op1.value != 0
                result << Term.new(op2, op1, :-)
                if op1.value != 1
                  result << Term.new(op2, op1, :'/') if op2.value %
                                                        op1.value == 0
                  result << Term.new(op1, op2, :*) if op2.value != 0 and
                                                      op2.value != 1
                end
              end
            end
          end
        end
      end

      result.each do | term | block.call(term) end
      memo[source] = result
    end
  end
end
```

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