Algorithms, A Dropbox Challenge And Dynamic Progra... Alan Skorkin Feb 27, 4:08 AM

Lately I've slowly been trying to grok the fullness of dynamic programming. It is an algorithmic technique that the vast majority of developers never master, which is unfortunate since **it can help you come up with viable solutions for seemingly intractable problems**. The issue with dynamic programming (*besides the totally misleading name*), is that it can be very difficult to see how to apply it to a particular problem and even when you do, it is a real pain to get it right. Anyway, I don't want to expound on this, I have something more interesting in mind.

The Dropbox Challenges

I was surfing the web the other day and in the course of my random wanderings I ended up at the Dropbox programming challenges page. Apparently, the Dropbox guys have posted up some coding challenges for people who want to apply to work there (*and everyone else, I guess, since it's on the internet and all :)*). Challenge 3 (***The Dropbox Diet***) immediately caught my eye since it looked like one of those problems that should have a dynamic programming solution, so I decided to use it as an opportunity to practice. The full description of the problem is on the challenge page, but here is the gist of it.

> We get a list of up to 50 activities and an associated calorie value for each (either positive or negative), we need to find a subset of activities where the sum of all the calorie values is zero.

It sounded easy enough until I thought about it and realised it was more complex than it first appeared. So, I went for a walk :) and when I came back I settled in to analyse it for real. **The first part of solving any problem is to really understand what problem you're trying to solve** (*that one sentence really deserves its own article*). In this case the activities list is just extraneous information, what we really have is a list of numbers and we need to find a subset of these numbers where the sum of the subset is equal to a particular value. It took me quite a while to come up with that definition, but once you have something like that, you can do some research and see if it is a known problem.

Of course, I did nothing of the kind, I had already decided that there must be a dynamic programming solution so I went ahead and tried to come up with it myself. This wasted about an hour at the end of which I had absolutely nothing; I guess my dynamic programming chops are still lamb-chops as opposed to nice meaty beef-chops :). Having failed I decided to do what I should have done in the first place – some research. Since I had taken the time to come up with a decent understanding of the problem, it only took 5 minutes of Googling to realise that I was dealing with the subset sum problem.

The Subset Sum Problem

The unfortunate thing about the subset sum problem is the fact that it's NP-complete. This means that if our input is big enough we may be in trouble. Wikipedia does give some algorithmic approaches to the problem (*no code though*), but just to cross our **t's** I also cracked open Cormen et al (*have you ever noticed how that book has everything when it comes to algorithms :)*). In this case the book agreed with Wikipedia, but once again, no code (*there are only two things I don't like about **Intro To Algorithms**, the lack of real code and the lack of examples*). I browsed the web some more, in case it would give me further insight into the problem, but there wasn't much more to know – it was time to get my code on.

The Exponential Time Algorithm

The problem with the exponential time algorithm is its runtime complexity (*obviously*), but our maximum input size was only 50 and even if that turned out to be too big, perhaps there were some easy optimizations to be made. Regardless I decided to tackle this one first, if nothing else it would immerse me in the problem. I'll demonstrate how it works via example. Let's say our input looks like this:

```
[1, -3, 2, 4]
```

We need to iterate through the values and on every iteration produce all the possible subsets that can be made with all the numbers we've looked at up until now. Here is how it looks:

Iteration 1:

```
[[1]]
```

Iteration 2:

```
[[1], [-3], [1, -3]]
```

Iteration 3:

```
[[1], [-3], [1, -3], [2], [1, 2], [-3, 2], [1, -3, 2]]
```

Iteration 4:

```
[[1], [-3], [1, -3], [2], [1, 2], [-3, 2], [1, -3, 2], [4], [1, 4], [-3, 4], [1, -3, 4],
```

On every iteration we simply take the number we're currently looking at as well as a clone of the list of all the subsets we have seen so far, we append the new number to all the subsets (*we also add the number itself to the list since it can also be a subset*) and then we concatenate this new list to the list of subsets that we generated on the previous iteration. Here is the previous example again, but demonstrating this approach:

Iteration 1:

```
[] + [1]
```

Iteration 2:

```
[1] + [-3], [1, -3]
```

Iteration 3:

```
[1], [-3], [1, -3] + [2], [1, 2], [-3, 2], [1, -3, 2]
```

Iteration 4:

```
[1], [-3], [1, -3], [2], [1, 2], [-3, 2], [1, -3, 2] + [4], [1, 4], [-3, 4], [1, -3, 4],
```

This allows us to generate all the possible subsets of our input, all we have to do then is pick out the subsets that sum up to the value we're looking for (*e.g. 0*).

The list of subsets grows exponentially (*it being an exponential time algorithm and all :)*), but since we know what sum we're looking for, there is one small optimization we can make. We can sort our input list before trying to generate the subsets, this way all the negative values will be first in the list. The implication here is this, once the sum of any subset exceeds the value we're looking for, we can instantly discard it since all subsequent values we can append to it will only make it bigger. Here is some code:

```
def subsets_with_sum_less_than_or_equal(reference_value, array)
  array = array.sort {|a,b| a <=> b}
  previous_sums = []
  array.each do |element|
    new_sums = []
    new_sums << [element] if element <= reference_value
    previous_sums.each do |previous_sum|
      current_sum = previous_sum + [ element ]
      new_sums << current_sum if current_sum.inject(0){|accumulator,value|accumulator+value} <
```

```
    end
    previous_sums = previous_sums + new_sums
  end
  previous_sums
end
```

If we execute that (*with our reference value being 0 and our array being [1, -3, 2, 4]*), we get the following output:

```
[[-3], [-3, 1], [-3, 2], [-3, 1, 2]]
```

All the subsets in that list sum up to less than or equal to our reference value (*0*). All we need to do now is pick out the ones that we're after.

```
def subsets_with_sums_equal(reference_value, array)
  subsets_with_sums_less_than_or_equal = subsets_with_sum_less_than_or_equal(reference_value,
  subsets_adding_up_to_reference_value = subsets_with_sums_less_than_or_equal.inject([]) do |a
    accumulator << subset if subset.inject(0){|sum, value| sum+value} == reference_value
    accumulator
  end
  subsets_adding_up_to_reference_value
end
```

This function calls the previous one and then picks out the subset we're after:

```
[[-3, 1, 2]]
```

It's simple and works very well for any input array with less than 20 values or so, and if you try it with more than 25 – good luck waiting for it to finish :). Exponential time is no good if we want it to work with an input size of 50 (*or more*) numbers.

The Dynamic Programming Algorithm

Both Wikipedia and Cormen tell us that there is a polynomial time approximate algorithm, but that's no good for us since we want the subsets that add up to exactly zero, not approximately zero. Fortunately, ***just like I suspected, there is a dynamic programming solution, Wikipedia even explains how it works, which is only marginally helpful when it comes to implementing it***. I know because that was the solution I tackled next. Here is how it works, using the same input as before:

```
[1, -3, 2, 4]
```

Just like with any dynamic programming problem, we need to produce a matrix, the key is to figure out what it's a matrix of (*how do we label the rows and how do we label the columns*). In this case the rows are simply the indexes of our input array; the columns are labelled with every possible sum that can be made out of the input numbers. In our case, the smallest sum we can make from our input is -3 since that's the only negative number we have, the biggest sum is seven (*1 + 2 + 4*). So, our uninitialized matrix looks like this:

```
+---+----+----+----+---+---+---+---+---+---+---+---+
|   | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---+----+----+----+---+---+---+---+---+---+---+---+
| 0 |    |    |    |   |   |   |   |   |   |   |   |
| 1 |    |    |    |   |   |   |   |   |   |   |   |
```

| 2 | | | | | | | | | | | |
| 3 | | | | | | | | | | | |

So far so good, but what should we put in every cell of our matrix. In this case every cell will contain either T (*true*) or F (*false*).

A T value in a cell means that the sum that the column is labelled with can be constructed using the input array numbers that are indexed by the current row label and the labels of all the previous rows we have already looked at. An F in a cell means the sum of the column label cannot be constructed. Let's try to fill in our matrix to see how this works.

We start with the first row, the number indexed by the row label is 1, there is only one sum that can be made using that number – 1. So only one cell gets a T in it, all the rest get an F.

	-3	-2	-1	0	1	2	3	4	5	6	7
0	F	F	F	F	T	F	F	F	F	F	F
1											
2											
3											

The number indexed by the second row label is -3, so in the second row, the column labelled by -3 will get a T in it. However, **we're considering the numbers indexed by the current row and all previous rows, which means any sum that can be made using the numbers 1 and -3 will get a T in its column**. This means that the column labelled with 1 gets a T and the column labelled with -2 gets a T since

```
1 + -3 = -2
```

	-3	-2	-1	0	1	2	3	4	5	6	7
0	F	F	F	F	T	F	F	F	F	F	F
1	T	T	F	F	T	F	F	F	F	F	F
2											
3											

We continue in the same vein for the next row, we're now looking at number 2 since it's indexed by the third row in our matrix. So, the column labelled by 2 will get a T, all the columns labelled by T in the previous row propagate their T value down, since all those sums are still valid. But we can produce a few other sums given the numbers at our disposal:

```
2 + -3 = -1
1 + 2 + -3 = 0
1 + 2 = 3
```

All those sums get a T in their column for this row.

	-3	-2	-1	0	1	2	3	4	5	6	7
0	F	F	F	F	T	F	F	F	F	F	F
1	T	T	F	F	T	F	F	F	F	F	F
2	T	T	T	T	T	T	T	F	F	F	F
3											

There are three patterns that are starting to emerge.

1. For every row, the column which is equivalent to the number indexed by the row get a T in it (*e.g. row zero represents the number 1, so the column labelled by 1 gets a T in row 0, row one represents the number -3 so the column labelled by -3 get a T in row 1 etc.*), every row will get one T in one of the columns via this pattern. **This is because a single number by itself is a valid subset sum**.

2. If a column already has a T in the previous row, this T propagates down to the current row (*e.g. when looking at the second row, the column labelled by 1 has a T in the first row and will therefore have a T in the second row also, when looking at the third row columns labelled by -3, -2 and 1 all had a T in the second row and will therefore contain a T in the third row*). **This is due to the fact that once it is possible to construct a certain sum using a subset of our input numbers, looking at more of the input numbers does not invalidate the existing subsets**.
3. Looking at any column label X in the current row which still has a value of F, if we subtract the number indexed by the current row from this column label we get a new number Y, we then check the row above the current row in the column labelled by Y, if we see a T, this T is propagated into the column X in the current row (*e.g. if we're looking at the second row, column labelled with -2, we subtract the number of the current row -3 from the column label, -2 – -3 = -2 + 3 = 1, this new number is the column label in the first row, we can see that in the first row in the column labelled with 1 there is a T, therefore this T gets propagated to the second row into the column labelled with -2*). **This is due to the fact that if we take a sum that is already possible and add another number to it, this obviously creates a new sum which is now possible**.

Those three patterns are the algorithm that we use to fill in our matrix one row at a time. We can now use them to fill in the last row. The number indexed by the last row is 4. Therefore in the last row, the column labelled by 4 will get a T (*via the first pattern*). All the columns that already have a T will have that T propagate to the last row (*via the second pattern*). This means the only columns with an F will be those labelled by 5, 6 and 7. However using pattern 3, if we subtract 4 from 5, 6 and 7 we get:

```
5 - 4 = 1
6 - 4 = 2
7 - 4 = 3
```

If we now look at the previous row in the columns labelled by those numbers we can see a T for all three cases, therefore, ***even the columns labelled with 5, 6 and 7 in the last row will pick up a T via the third pattern***. Our final matrix is:

```
+---+----+----+----+---+---+---+---+---+---+---+---+
|   | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
+---+----+----+----+---+---+---+---+---+---+---+---+
| 0 | F  | F  | F  | F | T | F | F | F | F | F | F |
| 1 | T  | T  | F  | F | T | F | F | F | F | F | F |
| 2 | T  | T  | T  | T | T | T | T | F | F | F | F |
| 3 | T  | T  | T  | T | T | T | T | T | T | T | T |
+---+----+----+----+---+---+---+---+---+---+---+---+
```

One final problem remains, how can we use this matrix to get the subset that adds up to the value we want (*i.e. 0*). This is also reasonably simple. We start in the column labelled by the sum we're after, in our case we start in the column labelled by zero. If this column does not contain a T then our sum is not possible and the input does not have a solution. In our case, the column does have a T so we're in business.

- We start at the last row in this column; if it has a T and the row above has a T we go to the row above.
- If the row above has an F then we take the number which is indexed by the current row and write it into our final output.
- We then subtract this number from the column label to get the next column label. We jump to the new column label and go up a row.
- Once again if there is a T there and there is an F above, then we write the number indexed by the row into our output and subtract it from the current column label to get the next column label.
- We then jump to that column and go up a row again.
- We keep doing this until we get to the top of the matrix, at this point the numbers we have written to the output will be our solution.

Let's do this for our matrix. We start at the column labelled by 0 since that's the sum we're looking for. We look at the last row and see a T, but there is also a T in the row above so we go up to that row. Now there is an F in the row above, so we write the number indexed by this row into our output:

```
output = [2]
```

We now subtract this number from our column label to get the new column label:

```
0 - 2 = -2
```

We jump to the column labelled by -2 and go up a row, there is another T there with an F in the row above, so we write the number indexed by the row to our output:

```
output = [2, -3]
```

We perform our subtraction step again:

```
-2 - -3 = -2 + 3 = 1
```

We now jump to the column labelled by 1 in the first row in the matrix. There is also a T there, so we need to write one last number to our output:

```
output = [2, -3, 1]
```

Since we're at the top of the matrix, we're done. As you can see the procedure we perform to reconstruct the output subset is actually a variant of the third pattern we used to construct the matrix. And that's all there is to it.

Oh yeah, I almost forgot the code :), since it is not tiny, I put it in a gist, you can find it here. But, here are the guts of it:

```
def initialize_first_row
  @matrix[1].each_with_index do |element,i|
    next if i == 0 # skipping the first one since it is the index into the array
    if @array[@matrix[1][0]] == @matrix[0][i] # the only sum we can have is the first numbe
      @matrix[1][i] = "T";
    end
  end
  @matrix
end

def populate
  (2...@matrix.size).each do |row|
    @matrix[row].each_with_index do |element,i|
      next if i == 0
      if @array[@matrix[row][0]] == @matrix[0][i] || @matrix[row-1][i] == &#39;T&#39; || cu
        @matrix[row][i] = "T";
      end
    end
  end
  @matrix
end

def current_sum_possible(row, column)
  column_sum = @matrix[0][column] - @array[@matrix[row][0]]
  column_index = @column_value_to_index[column_sum]
  return false unless column_index
  @matrix[row-1][column_index] == "T";
end

def derive_subset_for(reference_value)
  subset = []
  column_index = @column_value_to_index[reference_value]
  (1...@matrix.size).to_a.reverse.each do |row|
    if @matrix[row][column_index] == "F";
      return subset
    elsif @matrix[row-1][column_index] == "T";
      next
    else
      array_value = @array[row - 1] # the -1 is to account for the fact that our rows are 1
      subset.insert(0, array_value)
      column_index = @column_value_to_index[@matrix[0][column_index] - array_value]
    end
  end
  subset
end
```

You can recognise the 3 patterns being applied in the '*populate*' method. We're, of course, missing the code for instantiating the matrix in the first place. Grab the whole thing from the gist and give it a run, it generates random inputs of size 50 with values between -1000 and 1000. And if you think that would produce quite a large matrix, you would be right :) (*50 rows and about 25000 columns give or take a few thousand*). But **even with input size 100 it only takes a couple of seconds to get an answer, which is MUCH better than the exponential time algorithm**; in my book that equals success. Dropbox Challenge 3 – solved (*more or less :)*)!

By the way if you want to print out a few more matrices, grab the code and uncomment the relevant line (102) and you'll get a matrix similar to those above along with the rest of the output. Obviously, if you're doing that, make sure your input size is small enough for the matrix to actually fit on the screen. I used the great terminal-table gem to produce the nice ASCII tables.

Lastly, if you're wondering what framework this is:

```
if ENV["attest"]
  this_tests "generating subset sums using dynamic programming" do
    test("subset should be [1,-3,2]") do
      actual_subset_sum = subset_sum_dynamic([1, -3, 2, 4], 0)
      should_equal([1,-3,2], actual_subset_sum)
    end
    ...
  end
end
```

That would be me eating my own dog food, I took the time to write it, might as well use it :).

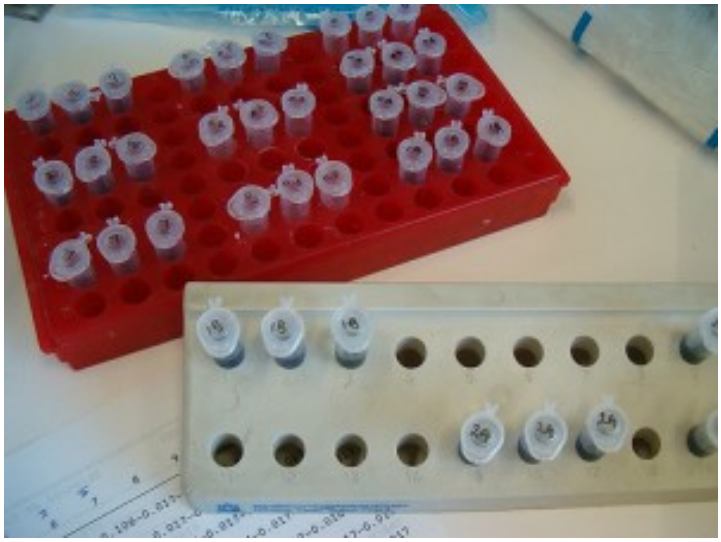
By the way, it took me hours (*pretty much the better part of a day*) to get all of this stuff working properly, dynamic programming algorithms really are fiddly little beasts. But, I had some fun, and got some good practice and learning out of it – time well spent (*and now there is some decent subset sum code on the internet :P*). Of course once I finished with this I had to look at the other challenges, number 2 didn't really catch my attention, but I couldn't walk away from number 1 with its ASCII boxes and bin packing goodness – I'll write that one up some other time.

Images by johntrainor, infinitewhite and familymwr

Related posts:

1. Sort Files Like A Master With The Linux Sort Command (Bash)
2. Output Redirection With Bash
3. A Wealth Of Ruby Loops And Iterators

A Unit Testing Framework In 44 Lines Of Ruby Alan Skorkin Feb 16, 11:58 AM



Late last year I attended some workshops which were being run as part of the YOW Melbourne developer conference. Since the workshops were run by @coreyhaines and @jbrains, TDD was a prominent part. Normally this would not be an issue, but in a spectacular display of fail (*considering it was a developer conference in 2010*), the internet was hard to come by, which left me and my freshly installed Linux laptop without the ability to acquire Rspec. Luckily a few weeks before, I decided to write a unit testing framework of my very own (*just because I could :)*) and so I had a reasonably fresh copy of that code lying around – problem solved. But, it got me thinking, **what is the minimum amount of code needed to make a viable unit testing framework**?

A Minimum Viable Unit Test

I had some minimal code when I first toyed with writing a unit testing framework, but then I went and ruined it by adding a bunch of features :), fortunately it is pretty easy to recreate. What we really want is the ability to execute the following code:

```
describe "some test" do
  it "should be true" do
    true.should == true
  end

  it "should show that an expression can be true" do
    (5 == 5).should == true
  end

  it "should be failing deliberately" do
    5.should == 6
  end
end
```

As you can see it looks very much like a basic Rspec test, let's try and write some code to run this.

Building A Basic Framework

The first thing we're going to need is the ability to define a new test using "*describe*". Since we want to be able to put a "*describe*" block anywhere (*e.g. its own file*), we're going to have to augment Ruby a little bit. **The "*puts*" method lives in the Kernel module and is therefore available anywhere (*since the Object class includes Kernel and every object in Ruby inherits from Object*), we will put describe inside Kernel to give it the same ability**:

```
module Kernel
  def describe(description, &block)
    tests = Dsl.new.parse(description, block)
    tests.execute
  end
end
```

As you can see, "*describe*" takes a description string and a block that will contain the rest of our test code. At this point, we want to pull apart the block that we're passing to "*describe*" to separate it into individual examples (*i.e. "it" blocks*). For this we create a class called **Dsl** and pass our block to its parse method, this will produce an object that will allow us to execute all our test, but let's not get ahead of ourselves. Our **Dsl** class looks like this:

```
class Dsl
  def initialize
    @tests = {}
  end
  def parse(description, block)
    self.instance_eval(&block)
    Executor.new(description, @tests)
```

```
  end
  def it(description, &block)
    @tests[description] = block
  end
end
```

What we do here is evaluate our block in the context of the current Dsl object:

```
self.instance_eval(&block)
```

Our **Dsl** object has an "*it*" method which also takes a description and a block and since that is exactly what our describe block contains everything works well (*i.e. we're essentially making several method calls to the "it" method each time passing in a description and a block*). **We could define other methods on our Dsl object and those would become part of the "*language*" which will be available to us in the "*describe*" block**.

Our "*it*" method will be called once for every "*it*" block in the describe block, every time that happens we simply take the block that was passed in and store it in hash keyed on the description. When we're done, we simply create a new **Executor** object which we will use to iterate over our test blocks, call them and produce some results. The executor looks like this:

```
class Executor
  def initialize(description, tests)
    @description = description
    @tests = tests
    @success_count = 0
    @failure_count = 0
  end
  def execute
    puts "#{@description}"
    @tests.each_pair do |name, block|
      print " - #{name}"
      result = self.instance_eval(&block)
      result ? @success_count += 1 : @failure_count += 1
      puts result ? " SUCCESS" : " FAILURE"
    end
    summary
  end
  def summary
    puts "\n#{@tests.keys.size} tests, #{@success_count} success, #{@failure_count} failure"
  end
end
```

Our executor code is reasonably simple. We print out the description of our "*describe*" block we then go through all the "*it*" blocks we have stored and evaluate them in the context of the executor object. In our case there is no special reason for this, but it does mean that the executor object can also be a container for some methods that can be used as a "*language*" inside "*it*" blocks (*i.e. part of our dsl can be defined as method on the executor*). For example, we could define the following method on our executor:

```
def should_be_five(x)
  5 == x
end
```

This method would then be available for us to use inside our "*it*" blocks, but for our simple test it is not necessary.

So, we evaluate our "*it*" blocks and store the result, which is simply going to be the return value of the last statement in the "*it*" block (*as per regular Ruby*). **In our case we want to make sure that the last statement always returns a boolean value (*to indicate success or failure of the test*), we can then use it to produce some meaningful output**.

We are missing one piece though, the "*should*" method, we had code like:

```
true.should == true
5.should == 5
```

It seems that every object has the "*should*" method available to it and that's exactly how it works:

```
class Object
  def should
    self
  end
end
```

The method doesn't really DO anything (*just returns the object*); it simply acts as syntactic sugar to make our tests read a bit better.

At this stage, we just take the result of evaluating the test, turn it into a string to indicate success or failure and print that out. **Along the way we keep track of the number of successes or failures so that we can produce a summary report in the end**. That's all the code we need, if we put it all together, we get the following 44 lines:

```
module Kernel
  def describe(description, &block)
    tests = Dsl.new.parse(description, block)
    tests.execute
  end
end
class Object
  def should
    self
  end
end
class Dsl
  def initialize
    @tests = {}
  end
  def parse(description, block)
    self.instance_eval(&block)
    Executor.new(description, @tests)
  end
  def it(description, &block)
    @tests[description] = block
  end
end
class Executor
  def initialize(description, tests)
    @description = description
    @tests = tests
    @success_count = 0
    @failure_count = 0
  end
  def execute
    puts "#{@description}"
    @tests.each_pair do |name, block|
      print " - #{name}"
      result = self.instance_eval(&block)
      result ? @success_count += 1 : @failure_count += 1
      puts result ? " SUCCESS" : " FAILURE"
    end
    summary
  end
  def summary
    puts "\n#{@tests.keys.size} tests, #{@success_count} success, #{@failure_count} failure"
  end
end
```

If we "*require*" this code and execute our original sample test, we get the following output:

```
some test
 - should be true SUCCESS
```

```
 - should show that an expression can be true SUCCESS
 - should be failing deliberately FAILURE
3 tests, 2 success, 1 failure
```

Nifty! Now, if you're ever stuck without a unit testing framework and you don't want to go cowboy, just spend 5 minutes and you can whip up something reasonably viable to tide you over. I am, of course, exaggerating slightly; you will quickly start to miss all the extra expectations, better output, mocking and stubbing etc. However **we can easily enhance our little framework with some of those features (*e.g. adding extra DSL elements*) – the development effort is surprisingly small**. If you don't believe me, have a look at bacon it is only a couple of hundred lines and is a reasonably complete little Rspec clone. Attest – the framework that I wrote is another decent example (*even if I do say so myself :P*). Both of them do still miss any sort of built-in test double support, but adding that is a story for another time.

Image by jepoirrier

Related posts:

1. True, False And Nil Objects In Ruby
2. Ruby Equality And Object Comparison
3. Using Ruby Blocks And Rolling Your Own Iterators

Read more…

The Greatest Developer Fallacy Or The Wisest Words Y... Alan Skorkin Feb 7, 11:42 AM

"*I will learn it when I need it*"! I've heard that phrase a lot over the years; it seems like a highly pragmatic attitude to foster when you're in an industry as fast-paced as software development. On some level it actually IS quite pragmatic, but on another level I am annoyed by the phrase. It has become a mantra for our whole industry which hasn't changed said industry for the better. The problem is this, **in the guise of sounding like a wise and practical developer, people use it as an excuse to coast**. There is too much stuff to know, it is necessary to be able to pick certain things up as you go along – part of the job. But, there is a difference between having to "pick up" some knowledge as you go along and doing absolutely everything just-in-time.

The whole industry has become a bunch of generalists, maybe it has always been this way, I just wasn't around to see it, either way I don't like it. Noone wants to invest the time to learn anything really deeply, not computer science fundamentals, not the latest tech you're working with, not even the language you've been coding in every day, for the last few years. Why bother, it will be replaced, superseded, marginalised and out of fashion before you're half way done. I've discussed this with various people many times, but noone seems to really see it as a problem. "*Just being pragmatic dude*". In the meantime we've all become clones of each other. You want a Java developer, I am a Java developer, you're a Java developer, my neighbour is a Java developer. What differentiates us from each other – not much! Well, I've got some jQuery experience. That's great, so you know how to build accordion menu then? Sure, I Google it and steal the best code I find :). In the meantime, if you need to hire a REAL expert (*in anything, maybe you're writing a fancy parser or need to visualise some big data*), I hope you've stocked up on beer and sandwiches cause you're gonna be here a while.

Ok, there are ways to differentiate yourself, I have better communication skills, which is why I do better. That's important too, but, **developers differentiating themselves based on soft skills rather than developer skills – seems a bit twisted**. We all communicate really well but the code is a mess :). Hell, I shouldn't really talk, I am a bit of a generalist too. Of course I'd like to think of myself as a T-shaped individual, but if we're completely honest, it's more of a dash-shaped or underscore-shaped with maybe a few bumps :). To the uninitiated those bumps might look like big giant stalactites – T-shaped indeed. **You seem**

like an expert without ever being an expert, just one advantage of being in a sea of generalists.

Investing In Your Future

I don't want to preach about how we should all be investing in our professional future, everybody knows we should be. Most people probably think they are infact investing, they rock up to work, write a lot of code maybe even do some reading on the side, surely that must make them an expert in about 10 years, and a senior expert in 20 (*I keep meaning to write more about this, one day I'll get around to it :)*)? But, if that was the way, every old person would be an expert in a whole bunch of stuff and that is emphatically not the case. Maybe it is just that people don't know how to build expertise (*there is an element of truth to this*), but I have a sneaking suspicion that **it's more about lack of desire rather than lack of knowledge**. What was that saying about the will and the way – totally applicable in this case?

I've gone completely off-track. "*Investing in professional future*" is just one of those buzzword things, the mantra is "*I will learn it when I need it*". It was good enough for my daddy and it has served me well so far. Let's apply this thinking to finance, "*I will invest my money when I think I need the money*". Somehow it doesn't quite have the same kind of pragmatic ring to it.

You Don't Know What You Don't Know

We've all had those moments where you're going through major pain trying to solve a problem until someone comes along and tells you about algorithm X or technology Y and it makes everything fast and simple. It was lucky that person just happened to be there to show you the "*easy*" way, otherwise you would have spent days/weeks trying to figure it out and it would have been a mess. You can't be blamed for this though, you don't know what you don't know. For me, this is where the "*I will learn it when I need it*" mentality falls over. **You can't learn something if you don't know it exists**. Google goes a long way towards mitigating this problem, but not all the way. There are plenty of problems you will encounter in the wild where you can beat your head against the wall ad infinitum unless you know what class of problem you're looking at (*e.g. if you know a bit about searching and constraint propagation, solving sudoku is easy, otherwise it's really quite hard*). You can't learn about an algorithm if you're not aware of it or its applicability. You can't utilise a technology to solve a problem if you don't even realise it has that capability. You're not going to always have someone there to point you in the right direction. I am willing to bet **there is a billion lines of code out there right now which can be replaced with a million lines of faster, cleaner, better code simply because whoever wrote it didn't know what they didn't know**.

I seem to be making a case for the opposite side here, if knowing what you don't know is the ticket then surely we should be focusing on breadth of knowledge. Superficial awareness of as much stuff as possible should see us through, we'll be able to recognise the problems when we see them and then learn what we need more deeply. Except it doesn't work like that, **skimming subjects doesn't allow you to retain anything**, our brain doesn't work that way. If we don't reinforce and dig deeper into the concepts we quickly page that information out as unimportant, it is a waste of time (*think back to cramming for exams, how much do you remember the next day?*). However if you focus on building deeper understanding of a subject – in an interesting twist – you will gain broad knowledge as well (*which you will actually be able to retain*). My grandad is a nuclear physicist, several decades of working to gain deeper knowledge of the subject has made him an expert, but it has also made him an excellent mathematician, a decent chemist, a pretty good geologist, a fair biologist etc. Just some empirical evidence that seeking depth leads to breadth as a side-effect.

Can You Learn It Fast Enough

Some stuff just takes a long time to learn. I am confident I can pick up an ORM framework I haven't seen before without even breaking stride, I've used them before, the concepts are the same. But what if you need to do some speech to text conversion,

not quite as simple, not enough background. Hopefully Google will have something for us to copy/paste. That was a bad example, only research boffins at universities need to do that crap. How about building a website then, we all know how to do that, but what if you need to do it for 10 million users a day. We just need to learn everything about scaling, **I am sure the users will wait a month or two for us to get up to speed :)**. Yeah, I am just being stupid, all we need to do is hire an expert and … errr … oh wait, we're all out of beer and sandwiches.

Why Should I Care

Working with experts is freaking awesome. You may have experienced it before, everything they say is something new and interesting, you learn new tricks with every line of code, you can almost feel your brain expanding :). You want to learn from the experts, so it's really sad when you can't find any. Since everyone is only learning when they "*need it*", noone can teach anything to anyone. The chunk of wisdom here is this, you want to work with experts, but the experts also want to work with experts, so **what are you doing to make sure the experts want to work with you**? Being able to learn something when you need it is a good skill to have, but you can not let it be your philosophy as a developer. Yes it is a big industry you can't learn everything, so pick something and make sure you know it backwards, if you're curious enough to follow up on the interesting bits, you'll find you have a decent grasp of a lot of other stuff at the end. And if you do a good enough job, other super-awesome-smart people are going to want to come and hang around you cause they'll be able to learn something from you and you'll be able to learn much from them. Everybody will be a winner.

Image by SamueleGhilardi and SpecialKRB

Related posts:

1. The Difference Between A Developer, A Programmer And A Computer Scientist
2. Here Are Some Words That Rhyme With Orange!
3. Building Software Development Expertise – Using The Dreyfus Model

Read more…

Here Is The Main Reason Why You Suck At Interviews Alan Skorkin Dec 8, '10, 12:11 PM

I've talked about interviews from one perspective or another on several occasions, you might even say it is a pet subject of mine. It's fascinating because **most people are no good at interviews** and when it comes to developer interviews – well; let's just say there is a whole new dimension for us to suck at with coding questions, whiteboards and whatnot. Of course, the other side of the equation is not pristine here, the interviewer can be just as much to blame for a terrible interview, either through lack of training, empathy, preparation or a host of other reasons, but that's a whole separate discussion. So, why are we so bad at interviews? You can probably think of quite a few reasons straight away:

- it is a high pressure situation, you were nervous
- you just didn't "*click*" with the interviewer
- you were asked all the "*wrong*" questions
- sometimes you just have a bad day

Infact, you can often work yourself into a self-righteous frenzy after a bad interview, thinking how every circumstance seemed to conspire against you, it was beyond your control, there was nothing you could do – hell, you didn't even want to work for that stupid company anyway! But, deep down, we all know that those excuses are just so much bullshit. The truth is there were many things we could have done, but by the time the interview started it was much too late. I'll use a story to demonstrate.

The least stressful exam I've ever had was a computing theory exam in the second year of my computer science degree. I never really got "*into it*" during the semester, but a few weeks before exam time – for some inexplicable reason – I suddenly found the subject fascinating. I spent hours reading and hours more playing with grammars and automata. Long story short, when exam time rolled around I knew the material backwards – I groked it. There was some anxiety (*you can't eliminate it fully*), but I went into the exam reasonably confident I'd blitz it (*which I did*). **Practice and preparation made all the**

difference. Of course, this is hardly a revelation, everyone knows that if you study and practice you'll do well (*your parents wouldn't shut up about it for years :)*). Interviews are no different from any other skill/subject in this respect, preparation and practice are key.

Can You Wing It?

The one difference with interviews is that they are an occasional skill, almost meaningless most of the time, but of paramount importance once in a long while. It just doesn't seem like it's worth the effort to get good at them, especially if you happen to already have a job at the time (*who knows, you may not need those skills for years*). There are plenty of other subjects clamouring for your attention and anyway every interview is different, you can never predict what's gonna happen so it would be stupid to waste your time trying. No – good communication skills and decent software development knowledge will see you through, right? Except, they don't and it won't. Sure, you might be able to stave off total disaster, but **without preparation and practice, you're mostly relying on luck**. Things "*click*"; you get asked the "*right*" questions and are able to think of decent answers just in time. This is how most people get hired. As soon as the process gets a little more rigorous/scientific, so many candidates get weeded out that companies like Google, Facebook, Twitter etc. find themselves trying to steal people from each other since they know that those that have passed the rigorous interview processes of their competitors must be alright. The interesting thing is that the rejected candidates are not necessarily worse; they are often simply a lot less prepared and a little less lucky.

Over the last few years presentation skills have seen quite a lot of press. Many a blog post and much literature has come out (*e.g. Presentation Zen and Confessions of a Public Speaker are both great books*). Many people have decent knowledge of their subject area and have good communication skills, they think they are excellent presenters – they are wrong. They put together some slides in a few hours and think their innate abilities will carry them through, but inevitably their presentations end up disjointed, mistargeted, boring or amateurish. Sometimes they sail through on luck, circumstances conspire and the presentation works, but these situations are not common. Malcolm Gladwell is a master presenter, he is one of the most highly sought after and highly paid speakers in the world (*and has written a bunch of awesome books to boot*) – this is not by chance. Without doubt he knows his stuff and has better communication skills than the majority of speakers out there and yet all his talks are rigorously prepared for and practiced. To my mind, **the situation with interviews is similar to that of presentations**, except the deluge of literature about interviews goes almost unnoticed since they are old-hat. The digital world hasn't changed the interview process too significantly (*unlike the public speaking process*), except the internet age brings all the old advice together in one place for us and all of that advice is still surprisingly relevant.

The Old-School Advice

Everyone (*and I mean everyone*) always says that you should research the company you'll be interviewing with beforehand. You would think people would have this one down by now, especially developers cause we're smart, right? Nope, no such luck, just about everyone who rocks up for an interview knows next to nothing about the company they are trying to get a job at, unless the company is famous, in which case people are just full of hearsay. But hearsay is no substitute for a bit of research and it is so easy, I am reminded of an article Shoemoney wrote about getting press (*well worth a read by the way, if you're trying to promote a product/service*) – there is a tremendous amount of info you can find out about a person by trawling the web for a bit and it is just as easy to learn something about a company. I mean, we do work in software, so any company you may want to work for should have a web presence and a half. And even if web info is sparse there is your social network or picking up the phone and seeing if someone will trade a coffee for some info. Yeah, you go to a bit of trouble but the fact that you did will be apparent in an interview, I mean, **there is a reason why I work where I work and I'd prefer to work with other people who give a shit** (*everyone would*) – you savvy? Of course if you do go to the trouble to find out what skills/tech/knowledge/processes a company is looking for/values you may be able to anticipate where an interview might head, the value there should be self-evident.

Which leads me to interview questions. When it comes to developers, there are three types of questions we struggle with/despise:

- the behavioural/culture fit
- the coding question
- the Mount Fuji question

With a bit of practice you can blitz all of these. The Fujis are the hardest, but even they can be prepared for, but I'll get back to those shortly.

Behavioural Questions

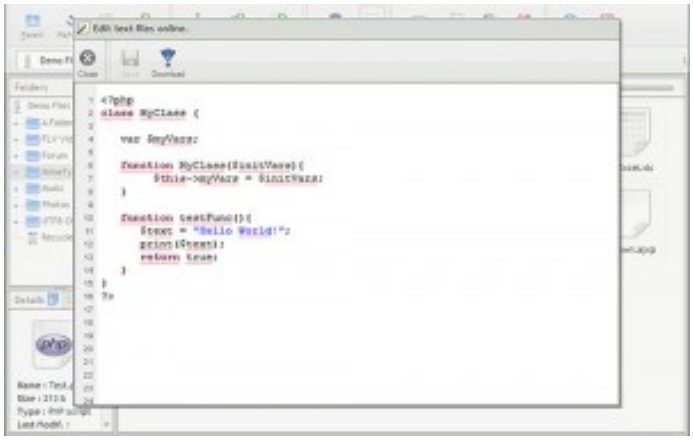
The behavioural questions seem annoyingly difficult but are actually the easiest. You know the ones "*Give me an example of a time when you demonstrated leadership/communication skills/problem solving/conflict resolution*". It is always the same question, with a different attribute of yours that you have to spruik. Being in essence the same question you can address all of

them with what amounts to the same answer, once again substituting a different attribute. These are actually difficult to handle on the spot, but if you have something prepared it is a breeze. Example:

> "There was a time, when Bill the developer was being an obstinate bastard and wouldn't buy in to the awesome that the rest of us were peddling, but I took charge of the situation and was able to convince Bill blah, blah …." – **leadership**
>
> "Bill the contrary developer just wouldn't agree with the way we were dishing out the awesome, but I took Bill out for coffee and we hashed it out one on one blah, blah …" – **communication**
>
> "There was insufficient awesome to go around and Bill and Joe just couldn't share it and were coming to blows, but I stepped in and took them both to a whiteboard, we had a long chat blah, blah …" – **conflict resolution**

As long as you think up a situation beforehand you can adapt it to answer any behavioural question, the situation can even be a fictitious, but you do need to think through it carefully for 10-15 minutes to be able to come up with something coherent. You will never have time to frame a coherent reply in the interview itself. Of course, it is best to have a few situations prepared just so you can change it up a bit, variety never hurt anyone. If the company you're interviewing with is enlightened they won't ask you these questions, but will instead focus on your dev skills, but there are many companies and few are enlightened, **might as well prepare for the worst case and be pleasantly surprised if the best case happens**.

Coding Questions



Talking about dev skills, the one thing that just about every company that hires developers will do, would be to ask a coding question at some stage. These usually take the form of a sandbox question or what I call a Uni question. You know the ones, "*Reverse a list*", "*Implement a linked list*" it's as if they are under the impression you studied computer science at some point, go figure :). **People struggle here, because you just don't come across this kind of question in your day-to-day work**. If they asked you to implement a Twitter or Facebook clone, you could really show them your chops, but balancing a binary tree – who the hell remembers how to do that? And that's the rub, you probably could dredge the information from the depths of your grey matter, but by the time you do the interview will have been long over. Because you don't do this kind of question daily, you brain has dumped the info you need to tape and sent it to Switzerland (*cause backups should be kept off-premises*). The answer is simple; you gotta practice these questions well in advance of the interview. Preparation baby, that's the ticket. Preferably you should be doing them regularly regardless of your employment status cause those questions are fun and bite-sized and give your brain a bit of a workout – you'll be a better developer for it. The most interesting thing though is this, you do enough of them and you won't really encounter anything new in an interview. There are really not so many themes when it comes to these questions, it will be the same formulae you'll just need to plug in different numbers (*a simplification but not too far from reality*). I really gotta follow my own advice here. If you seriously want a leg up, there are books specifically about this, I prefer Cracking the Coding Interview but there is also Programming Interviews Exposed – read them, do the questions.

Puzzles

Mount Fuji questions are the most controversial, but regardless of whether you hate them or not, even they can be practiced. Yeah, alright, maybe you're willing to walk out if any company ever dares to ask you about manhole covers, but **I'd much rather answer the questions and then walk out in self-righteous satisfaction rejecting the offers they'll be throwing at my feet, than storm out in self-righteous anger knowing deep down that I was a wuss for doing so**. And anyway, I'd like to think that I've already demonstrated that not all Mount Fuji questions are created equal, some are coding problems, others deal with concurrency, others still might be back-of-the-envelope questions (*the story of how these originated is actually pretty interesting, I am writing it up as we speak*). The subset of questions that are simply a useless puzzle are a lot smaller than you

might think. Doing these kinds of questions in your spare time is just another way to build your skills with a side benefit that you become an interview-proof individual. Of course, many people also enjoy an occasional puzzle – I'm just saying.

There is still lots more to say, I haven't even begun to talk about attitude, but this is already a **tl;dr** candidate, so I'll leave that discussion for another time. Let's sum up, **if you feel that you suck at interviews, it's because you didn't prepare well enough and haven't had enough practice** – it is that simple. As an interviewer it is most disheartening to see an unprepared candidate, there are just so many of them, on the other hand a person who is clearly on the ball is just awesome. And I am well aware that practicing interviews is hard, there is no Toastmasters equivalent for that particular skill, but thorough preparation can significantly mitigate that issue. Even a few minutes of preparation will put you head and shoulders above most other people since the vast majority don't bother at all. So, take the time to practice/prepare if you have an interview on the horizon, be smart about it and it is bound to pay off, not to mention a better experience for the interviewer as well, more fun and less stress for everyone.

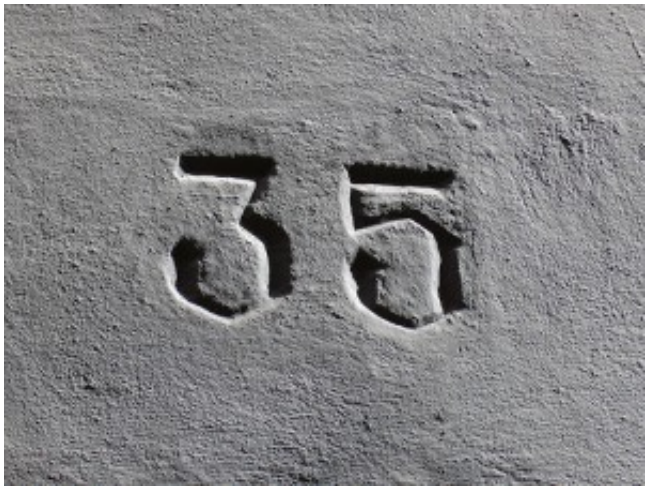
Images by Caro Wallis and louisvolant

Related posts:

1. The Best Way To Interview A Developer
2. How To Answer A Programming Interview Question And Look Good Doing It
3. When A 'Mount Fuji' Question Is Not Really A 'Mount Fuji' Question (Are You As Smart As You Think You Are)

Read more…

Converting Integers To Words – Bringing Order To E... Alan Skorkin Nov 5, '10, 2:58 PM



For a few years now, I've been meaning to read Peter Norvig's old book, "*Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*", but every time I started I'd inevitably get lost in the code due to my poor Lisp skills. For some reason I just couldn't absorb the introductory Lisp chapters at the beginning of the book. So a little while ago I decided to "*begin at the beginning*" and slowly started making my way through Peter Seibel's "*Practical Common Lisp*".

Inevitably, one of the first things I learned about was the "*format*" function. *Format*, is essentially the Lisp equivalent of *printf*, so you kinda need it to produce the old "*Hello World*". Just like *printf* and its ilk, *format* supports string interpolation via embedded directives. For example if you wanted to print two strings side by side but separate them by a specific number of spaces you could do something like this:

```
CL-USER> (format t "~a~40t~a~%" "abc" "123")
abc                                     123
NIL
```

One of the most interesting format directives is ~r which will take an integer and produce its English representation e.g.:

```
CL-USER> (format nil "~r" 123456)
"one hundred twenty-three thousand four hundred fifty-six"
```

I found this little gem in one of the footnotes, the lesson here being – **always read the footnotes**. Anyway, I don't know about you, but I thought that was pretty awesome, which immediately got me thinking about how hard it would be to implement a stand-alone integer to English translation function. I would have loved to try my hand at doing this in Lisp, but I don't think I am sufficiently Jedi for that as yet (*I will however give it a go as soon as I am ready, you will not want to miss* ***that*** *:)*), so instead I decided to have a go at implementing it in Ruby.

A Totally Unrelated Musical Trivia Interlude

Speaking of Jedi, I've recently been a little obsessed with the "*Good Morning*" song from "*Singing In The Rain*", you know the one I mean. I blame "*Family Guy*" for this, it all started after I watched the episode where they perform this song in one of their postmodern flashes of randomness. Anyway, it's just such a nice and happy song, it's kinda quaint how 1:30 used to be considered "*late*" in those days and I dig the whole synchronized tap dancing thing that they do (*just imagine how much they had to practice to get it right*).

Now, here is the trivia, the girl in the song/movie is a 20 year old Debbie Reynolds (*did you know that before "Singing In The Rain" she didn't even know how to dance*) who went on to marry singer Eddie Fisher. A few years later she gave birth to a daughter named Carrie (i.e. Carrie Fisher) who, in turn, went on to play Princess Leia in "*Star Wars*". That's how all of this relates to Jedi, and you thought I was going off on a random tangent – now you know better :). Time to get back to some Ruby code.

A Ruby Based Integer To English Converter

It turns out writing it was somewhat harder than I expected because **the English language is a little insane when it comes to naming numbers**. Everything was going well at the start; it was obvious that we need some kind of lookup table to convert digits to words. The lookup table must contain all the uniquely named numbers that we are likely to encounter.

- all number between 1 and 10 (*e.g. one, two, five etc.*)
- all number between 11 and 20 (*e.g. eleven, twelve, sixteen etc.*)
- all multiples of 10 between 20 and 100 (*e.g. thirty, forty etc.*)
- all uniquely named powers of 2 over 100 (*e.g. hundred, thousand, million, billion etc.*)

Here is a cut-down version:

```
{
1 => "one",
2 => "two",
3 => "three",
...
19 => "nineteen",
20 => "twenty",
...
90 => "ninety",
100 => "hundred",
1000 => "thousand",
1000000 => "million",
...
}
```

We don't really think about it, since we are so used to it, but that is actually a lot more unique numbers than strictly necessary. **You can easily identify all possible numbers using just 1 through 9 as well as all the named powers of 10**, such as ten, hundred, thousand etc (*we'll explore this thought further shortly*). Doing it this way would be pretty logical, instead the English speaking world decided that we needed nine special words to represent numbers between 11 and 19 as well as eight more similar, but still unique, words for the multiples of 10 between 20 and 100. As much as we take it for granted this causes all sorts of pain when we try to implement out integer to words converter.

Consider this, we're given an integer several digits long, let's say 12.

- to get a word representation we start looking at the digits of the integer in reverse order (*we have to start with the least significant digit first*):
  - digits = 21
- we use the look-up table to find the word that represents the first digit and add it to an accumulator (*which is a list that will contain all the words in the English representation of our integer*):
  - digits = **2**1, accumulator = [**two**]

But as soon as we move on to the second digit, we're already in trouble. If this digit is a one, we know we have a number between 11 and 19, so we have to do the following:

- backtrack the previous digit from the accumulator:
  - digits = 2**1**, accumulator = []
- get the current digit along with the previous digit and reverse them to get our actual number:
  - digits = **12**, accumulator = []

- do a lookup to get the new word representation and put the new representation in the accumulator:
    - digits = **12**, accumulator = [**twelve**]

The story is similar if the second digit is 2 through 9, we still have to backtrack, then we need to do a lookup on the second digit multiplied by 10 (i.e. 20, 30 etc), combine the result with the word representation of the previous digit and put that back in the accumulator. So if the number is 20, we need to look up 20, combine that with 5 and get twenty five which ends up in the accumulator.

When the number is big enough, this patters comes up again and again every few digits. For example, numbers such as 11000, 23000, 134000 need to be represented as **eleven** thousand, **twenty three** thousand and one hundred and **thirty four** thousand (*see what I mean*). Luckily it is a pattern so we can handle all occurrences of it as a special case, but it's a pattern that need not be. And talking about patterns that need not be, what is our obsession with the word hundred:

- three **hundred** and fifty
- one **hundred** and twenty thousand three **hundred** and fifty
- one **hundred** and twenty five million three **hundred** and twenty thousand five **hundred** and six

We have the word hundred appearing 3 times in that last one, we're used to it and it feels comfortable, but is it strictly necessary? We have to handle that one as a special case as well. Needless to say, the code ends up being long and ugly, you can have a look at the full source here, but the main chunk of it is something along the following lines:

```
def convert(number)
  return "zero" if number == 0
  word_representation_accumulator = []
  number_digits_reversed = number.to_s.reverse
  digit_count = 0
  number_digits_reversed.chars.each_with_index do |digit, index|
    digit_as_number = Integer(digit)
    skip_zero(digit_as_number) do
      if digit_count == 0
        word_representation_accumulator << "#{NTW[digit_as_number]}"
      elsif ten_to_twenty?(digit_as_number, digit_count)
        backtrack word_representation_accumulator
        actual_number = Integer("#{digit}#{number_digits_reversed[index - 1]}")
        multiplier = (digit_count > 1 ? 10**(digit_count - 1) : nil)
        word_representation = "#{NTW[actual_number]}"
        word_representation += " #{NTW[multiplier]}" if multiplier
        word_representation += " and" if word_representation_accumulator.size == 1
        word_representation_accumulator << word_representation
      elsif twenty_to_one_hundred?(digit_count)
        backtrack word_representation_accumulator
        multiplier = (digit_count > 1 ? 10**(digit_count - 1) : nil)
        lookup_number = digit_as_number * 10
        word_representation = "#{NTW[lookup_number]}"
        word_representation += " #{NTW[Integer(number_digits_reversed[index - 1])]}"
        word_representation += " #{NTW[multiplier]}" if multiplier
        word_representation += " and" if word_representation_accumulator.size == 1
        word_representation_accumulator << word_representation
      elsif digit_count == 2 || digit_count % 3 == 2
        multiplier = 10**2
        word_representation = "#{NTW[digit_as_number]} #{NTW[multiplier]}"
        word_representation += " and" if word_representation_accumulator.size != 0
        word_representation_accumulator << word_representation
      else
        multiplier = 10**digit_count
        word_representation = "#{NTW[digit_as_number]} #{NTW[multiplier]}"
        word_representation += " and" if word_representation_accumulator.size == 1
        word_representation_accumulator << word_representation
      end
    end
    digit_count += 1
  end
```

```
    word_representation_accumulator.reverse.join(" ")
  end
```

You can have a go at running it, but don't forget, I only took the lookup table up to trillions, so if you want to try numbers bigger than that you will need to add more named powers of ten (*such as quadrillion, quintillion etc.*) to the table. Here is some sample output:

```
315119 - three hundred and fifteen thousand one hundred and nineteen
1000001 - one million and one
1315119 - one million three hundred and fifteen thousand one hundred and nineteen
11315119 - eleven million three hundred and fifteen thousand one hundred and nineteen
```

Using Code To Derive The English That Should Have Been

Even though my code was producing reasonable output, I wasn't entirely happy, the arbitrary nature of the English language made my code uglier than it could have been. Since I've taken liberties with the English language before I decided this was a good time to do it again. What if we were to remove all the extraneous words that English had for numbers. Surely, the numbers 1 through 9 are more than sufficient? This would be the simplest possible case e.g.:

```
135 - one three five
3619 - three six one nine
```

Unfortunately it just doesn't cut it. Firstly we would have to introduce 0 as a non-silent number e.g.:

```
609 - six zero nine
```

Currently, zero is completely silent in all numbers unless it is by itself e.g.:

```
609 - six hundred and nine (see, zero is silent)
```

But even if this wasn't an issue, it turns out **we actually need the named powers of ten to make English representations of integers meaningful**. *One five six two three*, tells us almost nothing, but *fifteen thousand six hundred and twenty three* gives us a lot of information instantly. Even when we haven't processed the number fully, we know straight away that it is approximately *15 thousand and a few hundred*. The bigger the number gets, the more pronounced this "*estimation*" effect is. So, we need numbers one through nine as well as named powers of 10. This allows us to cut our lookup table down to the following:

```
NTW = {
1 => "one",
2 => "two",
3 => "three",
4 => "four",
5 => "five",
6 => "six",
7 => "seven",
8 => "eight",
9 => "nine",
10 => "ten",
100 => "hundred",
1000 => "thousand",
1000000 => "million",
1000000000 => "billion",
1000000000000 => "trillion"
}
```

We can use this table to write code that is much more intuitive than our previous attempt. Infact, we don't even need to know exactly what we're trying to achieve, instead we let the code guide us to the "*correct*" solution. We're using code to derive the requirements for the real world which is the opposite of the usual scenario. The full implementation I came up with is in this gist, but here is the main bit:

```
def convert(number)
  return "zero" if number == 0
  word_representation_accumulator = []
```

```
    number_digits_reversed = number.to_s.reverse
    digit_count = 0
    number_digits_reversed.chars.each_with_index do |digit, index|
      digit_as_number = Integer(digit)
      skip_zero(digit_as_number) do
        multiplier = 10**digit_count
        word_representation = "#{NTW[digit_as_number]}"
        word_representation += " #{NTW[multiplier]}" if multiplier > 1 && NTW[multiplier]
        word_representation_accumulator << word_representation
      end
      digit_count += 1
    end
    word_representation_accumulator.reverse.join(" ")
  end
```

You have to admit, it's much shorter and nicer looking and here is some of the output produced:

```
0 - zero
1 - one
3 - three
5 - five
11 - one ten one
15 - one ten five
25 - two ten five
71 - seven ten one
40 - four ten
100 - one hundred
101 - one hundred one
112 - one hundred one ten two
123 - one hundred two ten three
457 - four hundred five ten seven
999 - nine hundred nine ten nine
1000 - one thousand
1001 - one thousand one
1010 - one thousand one ten
1011 - one thousand one ten one
2117 - two thousand one hundred one ten seven
3001 - three thousand one
13101 - one three thousand one hundred one
14001 - one four thousand one
16000 - one six thousand
25119 - two five thousand one hundred one ten nine
65009 - six five thousand nine
315119 - three one five thousand one hundred one ten nine
1000001 - one million one
1315119 - one million three one five thousand one hundred one ten nine
11315119 - one one million three one five thousand one hundred one ten nine
74315119 - seven four million three one five thousand one hundred one ten nine
174315119 - one seven four million three one five thousand one hundred one ten nine
1174315119 - one billion one seven four million three one five thousand one hundred one ten ni
15174315119 - one five billion one seven four million three one five thousand one hundred one
35174315119 - three five billion one seven four million three one five thousand one hundred or
935174315119 - nine three five billion one seven four million three one five thousand one hunc
2935174315119 - two trillion nine three five billion one seven four million three one five thc
```

As you can see, all the named powers of ten are now treated in exactly the same way i.e.:

- 10 is ten to the power of 1
- 100 is ten to the power of 2
- etc.

So 101 is **one hundred one** and 11 is **one ten one**, 203 is **two hundred three**, while 23 is **two ten three** – logical isn't it?

There is no longer any need for any specially named numbers between 10 and 100 since we can represent all of them just as efficiently with this notation; this means no more special cases for numbers under 100. The other one we had was all the extra usages of the word hundred. We simply eliminate them all together (*except for when we are talking about numbers between 100 and 1000*), **three one five thousand** conveys the same amount of information as **three hundred and fifteen thousand** – the word hundred is totally redundant. Now, every named power of 10 appears only once in the English representation of the integer which allows us to still retain the ability to "*estimate*" the magnitude instantly without any extraneous information:

```
one million three one five thousand one hundred one ten nine
```

Short, consistent and to the point. It does seem a little awkward at first, but after a few minutes it starts to grow on you – after all, **it is much more logical than the system we have now**. There you go, deriving "*better*" English through some Ruby code. Not only do we get superior language but it is also easier, faster and less error prone to code it – FTW brothers, FTW :)!!!

Image by plindberg

Related posts:

1. Tweaking English For Fun And Profit … Facilitating Poetry
2. Here Are Some Words That Rhyme With Orange!
3. Timing Ruby Code – It Is Easy With Benchmark

Write A Function To Determine If A Number Is A Po... Alan Skorkin Oct 18, '10, 12:59 PM

One of my friends always asks that question when interviewing developers. Apparently it's quite ambiguous and many people misunderstand – which is curious since I thought it was rather straight forward, but that's not what this story is about.

When he first told me about this question my brain immediately went ahead and tried to solve it, as your brain probably did as soon as you read the title of this post, if you're a developer :). After thinking about it for a couple of minutes I said that **people should get "*extra points*" if their function uses bit hackery to solve the problem.** I later realised that the most interesting thing about this was how I arrived at that conclusion.

The First Thing I Thought Of Was…

Something along the lines of the following:

```
def power_of_2?(number)
 return false if number == 0
 while(number % 2 == 0)
   number = number / 2
 end
 return false if number > 1
 true
end
```

Of course it didn't spring forth out of my head fully formed like that, but the general gist was the same. This code solves the problem and, as you can see it is an iterative solution. The first programming language I learned was Java, the second was C, so you might say I was weaned on the "*iterative approach*". So, whenever I am presented with a new problem, my mind always gropes for the iterative solution first. You might say I find it the most "*natural*" way to solve a problem.

The Second Thing I Though Of Was…

Something that looked like this:

```
def power_of_2?(number)
 return true if number == 1
 return false if number == 0 || number % 2 != 0
```

```
 power_of_2?(number / 2)
end
```

Once again, it solves the problem, but this time recursively. Since those early days of Java and C, I've played around with a whole bunch of languages, some just as imperative, but others were functional or had functional capabilities. In such languages, recursion is the norm or at least a lot more prevalent. I find that I really like recursive solutions, it is often a very elegant way to get around some ugly code. Even though it is not the first stop my brain makes when it comes to solving programming problems, I find that these days I almost always consider if there might be a recursive way to approach a problem. I don't yet always instinctively feel if a problem has a recursive answer, but in many cases (*like this one*) it is fairly simple. As long as you can divide the initial problem into two smaller parts which are "*equivalent*" to the original problem, you can solve it recursively (*like when you split a tree you get two smaller trees*).

I do wonder if programmers who were "*weaned*" on functional languages would go for the recursive solution first and if they find one do they bother to consider an iterative one at all? If you're such a developer, please share how you think.

The Next Thing I Thought Of Wasn't Code

It is only recently that I have started to give bit hacking the attention it deserves. It always used to seem like it wasn't worth the trouble, but since I've learned a bit more about Bloom Filters (I'll write it up one of these days, hopefully make it a bit easier to grok) and finally got my hands on a copy of "Programming Pearls", I've become a bit of a convert. So, whenever I hear "*power of 2*" or anything similar these days, I always think that there is surely some bit hacking to be done :), which is exactly what my brain latched on to after the iterative and recursive approaches.

Consider that any number that is a power of 2 has exactly one bit set in its binary representation e.g.:

```
2    =  00000010
4    =  00000100
8    =  00001000
16   =  00010000
256  = 100000000
```

If we subtract 1 from any of these numbers we get the following:

```
2 - 1 = 1 = 00000001
4 - 1 = 3 = 00000011
8 - 1 = 7 = 00000111
```

Every bit that was less significant than the original interesting bit (*the one that was set*), is now set while the original bit is unset. If we now do a bitwise and (*& operator*) on the original number and the number which results when 1 is subtracted, we always get zero:

```
2 & 1 = 00000010 & 00000001 = 0
8 & 7 = 00001000 & 00000111 = 0
```

This will only happen if a number is a power of two. Therefore we can write the following code:

```
def power_of_2?(number)
 number != 0 && number & (number - 1) == 0
end
```

This is apparently a pretty well known way to determine if a number is a power of 2. **In the olden days when computing resources were scarce, every programmer probably knew about this shortcut and others like it, these days – not so much**. But, you have to admit it is by far the cleanest solution of the three, even the need to treat zero separately makes it only marginally less elegant. Definitely a handy thing to remember (*it does still come up occasionally in the real world*).

I do wonder though, is anyone's brain sufficiently indoctrinated into bit hackery that they would jump to this solution first without even considering an iterative or a recursive one? Anyone?

If Someone Ever Actually Asks You This In An Interview…

The first thing to do would be to write (*or at least mention*) some tests. They don't have to be proper unit tests in this case (*although for more complex questions it may be warranted*), but you will never go wrong if you show that you're aware of testing. In this case something along the lines of this is probably fine:

```
puts "fail 2" if !power_of_2? 2
puts "fail 4" if !power_of_2? 4
puts "fail 1024" if !power_of_2? 1024
puts "fail 1" if !power_of_2? 1
puts "fail 1025" if power_of_2? 1025
puts "fail 0" if power_of_2? 0
puts "fail 3" if power_of_2? 3
```

0,1,2 and 3 are the cases where our function is likeliest to break the rest are included for completeness. The other thing that may be worth mentioning is the fact that the behaviour of the function for negative integers or floats is undefined, you can of course handle those cases, but it adds extra complexity and I think simply showing that you're aware of them is enough.

After this you can go ahead and write your function, remember – bit hackery means extra points (*as far as I am concerned anyway :)*).

Image by .m for matthijs

Related posts:

1. Solving The Towers Of Hanoi Mathematically And Programmatically – The Value Of Recursion
2. Are You Using The Full Power Of Spring When Injecting Your Dependencies?
3. How To Write A Simple Web Crawler In Ruby

Read more…

---

99 Out Of 100 Programmers Can't Program – I Call B... Alan Skorkin Oct 16, '10, 2:27 PM

Something has always struck me as a little bit off, about that "statistic" (*or its equally unlikely brothers, 199 out of 200, 19 out of 20 etc.*). I remember reading Joel's post alluding to this back in the day, then Jeff's a couple of years ago, there were a few others, most recently this one. And as much as I want to just accept it (*for reasons of self-aggrandisement*), I can't. I've been in this industry for a few years now, in that time, I've met some really good developers, a bunch of average ones, even the odd crappy one, but **I am yet to meet the veritable army of totally useless non-programming programmers** that must surely exist if those numbers are accurate (*the "architects" don't count :P*).

Why Should Top Developers Seek You Out?

Whenever I see the latest post about how hard it is to hire good people, because x out of y developers are useless, one question immediately springs to mind. **Is that x out of y applicants, or x out of y working developers?** There is a massive distinction. Unless you're a company trying to compile stats by doing an industry-wide study, you can't really comment on the skill levels of all working programmers in any authoritative fashion. So, we must be talking about x out of y applicants. But once again, it's not applicants in general it's applicants to YOUR company. All of a sudden the headline is:

> **X out of Y Applicants for Positions Advertised By My Company Can't Program**

That's a whole lot less impressive/sensational and whole lot closer to reality. But, let us dig a little deeper. It's not just the one company, too many other people/companies have had the same pain. Which is perhaps why these posts receive so much attention. It's cathartic to have a bit of a whinge along with a bunch of other people who feel your pain :). My question for everyone is this, what makes your company so special? How much time did you spend making sure your ad was sufficiently attractive to the star developers and a sufficient deterrent to the crappy ones? I can tell you right now the fact that you're the "*world's leading enterprise wodget provider*", the latest "*well-funded social startup*" or pay "*above industry rates*", is not going to bring the coding elite knocking on your door. I guess it comes down to this, **if you try to attract talent in a generic fashion you will attract a generic response**, meaning that there is a decent chance your 200 applicants were ALL a bunch of discards.

Blame Your Interview Process First

Go back to that article of Joel's that I mentioned above where he talks about the 199 useless programmers who apply for every job thereby inflating the applicant numbers. I don't think those 199 wannabe programmers exist, I think the pool is much larger. There is a whole bunch of, let's call them "*aspiring programmers*" who are totally crap and either can't get a job or can't keep one, which doesn't stop the from trying. A significant percentage of your applicants are those guys and yeah they can't program, but then again I wouldn't really call them "*programmers*" either. It wouldn't take long to get discouraged and cynical about the whole industry when dealing with those guys for days on end. But let's say you can screen all those out via resumes or whatever and only end up with seemingly legitimate applicants, how come so many of those can't code their way out of a paper bag?

Firstly, you didn't really screen all the discards out via the resumes, that's not possible, **I've seen some highly impressive resumes from some highly unimpressive people**. All our stats are already suspect at this point, but let's plow on anyway. Likely the next course of action is to screen further via phone or face-to-face or both. We ask simple coding questions like the fizzbuzz, but our applicants still fail – even ones that shouldn't:

> "Here is the question that the vast majority of candidates are unable to successfully solve, even in half an hour, even with a lot of nudging in the right direction:
>
> *Write a C function that reverses a singly-linked list.*
>
> That's it. We've turned away people with incredibly impressive resumes (including kernel developers, compiler designers, and many a Ph.D. candidate)…"

That's from the RethinkDB post that I also mentioned above (I *am not picking on the RethinkDB guys, it is just conveniently the latest post on the subject that I have read :)*). Kernel developers, and Ph.Ds can't reverse a list? That seems entirely unlikely. Perhaps it is not the people who are to blame but the process. We have learned, especially over the last few years that it is often the process that prevents a software team from being productive. These days most teams would cast a critical eye towards their process when looking for causes of dysfunction, before they start pointing fingers at each other. So, why not cast the same critical eye towards the interview process? Perhaps the objectives of the interview are unclear, or you're not communicating well enough, or you're using the wrong medium for what you're trying to achieve (*e.g. coding over the phone*), or you didn't prepare thoroughly enough as an interviewer added to the probable unpreparedness of the interviewee (*this chronic unpreparedness is endemic in our industry and deserves a post of it's own*). Sounds like the same kind of issues that cause software projects to go off the rails :). My point is, **it is not necessarily the fact that the candidate is crappy it could be that you're just doing it wrong**.

Just Because It Makes Us Feel Good Doesn't Make It True

As I was thinking about all this stuff I found myself referring to the X out of Y "*statistics*" as "*feel-good numbers*" because they make us all feel good about ourselves. I mean, it's pretty sad for the 99 out of 100 poor schlubs, but not you and me – we are coding ninjas. How does it feel to be special :)? Somehow though I don't reckon there are 99 working programmers sitting there reading those posts thinking "*…yeah I am a bit of a failure, I wish I was one of those 1 in a 100 dudes*". As a decent programmer, look around yourself. **Can you really honestly say the vast majority of the developers you're working with have trouble with loops or basic arithmetic**? If you can, I fail to see how your company can produce any kind of working software and why are you hanging around that place anyway, it surely is not healthy for your career not to mention your sanity.

If you've read anything I've written before, you know that I am not one to shy away from a generalisation, but in this case I believe it gives a skewed picture that needlessly makes the whole industry look bad. There is no denying that lack of skill can be a problem in software, but then again this is one of the few professional disciplines where you can read a book and "*wham bam thank you ma'am*", you're a programmer, or at least think that you are (*and might even be able to find work if someone is desperate/stupid enough and you're smooth enough*). Try doing the same thing in the medical profession or law, or accounting. This is one of the problems that is unique to IT (*other industries have their own*), we deal with it to the best of our ability. This issue will always make the hiring process difficult, which makes it doubly as important to think long and hard about how to attract decent people and tweak your interview process to make sure you're getting what you need out of it. And yes it will be time consuming and difficult and will make you feel like you're wasting time instead of doing "*real work*", but then again you know what the other side of the coin is.

Image by nitot

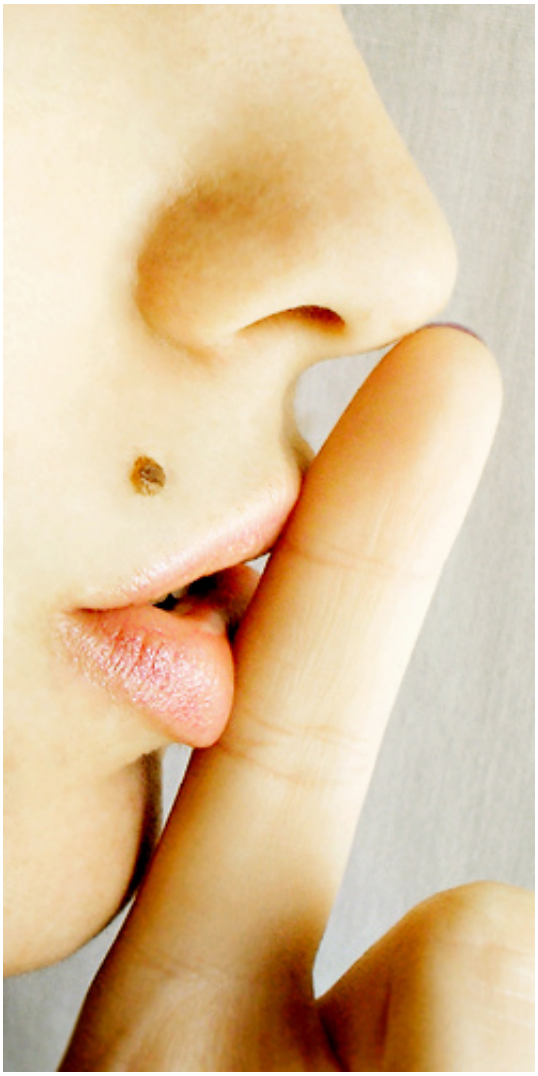
Related posts:

1. The Best Way To Interview A Developer
2. High Academic Results Make Better Programmers

3. How To Be A Real Elite Programmer And Make Sure Everybody Knows It

Read more…

Learning A Software Development Lesson From A C... Alan Skorkin Jun 7, '10, 12:33 PM



As I was getting ready for work the other day, some children's program was on TV. Normally it's just background noise, but today something made me pay attention. Here is what I heard:

A wise old owl sat in an oak,
The more he heard, the less he spoke,
The less he spoke, the more he heard,
Why aren't we all like that wise old bird.

I've recently been on a kick of extracting software development related lessons from everyday situations and events – that little poem immediately made me think of opinions. Software developers are an opinionated bunch, which is fair enough – comes with the territory. **The trick with opinions though, is knowing when to shut up and listen to those of others**. I could probably stand to follow that particular advice a lot more often :). If you can master that trick you're guaranteed to learn something. Even if all you learn is how to keep your opinion to yourself, once in a while, that's still a skill worth having.

Having said all of that, I do believe that you need to form an opinion about everything that happens, you don't have to defend your opinion to the death, but you do need to have one. There are important decisions to be made in software projects every day, regarding technology, process, requirements etc. You're not always going to be the most qualified person to make a decision, but if you let things slide without expressing your point of view – you deserve everything that happens. In that, software is like government. ***Having an opinion is the first step towards taking your destiny as a developer into your own hands*** (*this applies equally to teams*). And that's all I have to say about that for now.

Image by NuageDeNuit

Related posts:

1. Does Software Development Have A Culture Of Rewarding Failure
2. On The Value Of Fundamentals In Software Development
3. Software Development And The Sunk Cost Fallacy

Read more…

When A 'Mount Fuji' Question Is Not Really A 'Mount ... Alan Skorkin Jun 2, '10, 2:45 PM

Many people hate '*Mount Fuji*' style interview questions. Questions such as:

"*How many barbers are there in your home town?*"

or

"*How would you move Mount Fuji?*"

That last one being the most well known of these types of questions and the one from which the rest get their name. **These questions were initially popularized by Microsoft**, but many of the most well-known tech companies have used them since. They have recently fallen a bit out of favour, but you're still likely to come across some if you do few tech interviews here and there.

There is a good reason why many people don't like these types of questions. Most of the time, they don't really have a good answer. The question is designed to put a person under pressure (*when they can't immediately see a way to tackle the problem*) and see how they handle it, as well as see if they can come up with a creative solution on the spot. Some developers tend to enjoy that kind of challenge, others seem to loath it which is why opinions about these questions are highly polarised. While **I personally don't think these questions are the be-all-end-all, I do believe they have their value**, but that is not what I want to talk about.

You see the people who hate these questions, hate them with a passion. Whenever an interview contains a '*Mount Fuji*' or two and people do badly, they know exactly where to cast the blame. "*It was all because of the stupid Mount Fuji questions. I was never good at those, plus I wouldn't want to work for a company that would ask them anyway.*" That's the excuse you give to yourself. Makes you feel better about failing – which is actually not a bad rationalisation as far as rationalisations go. But, there is an issue with this thinking because **sometimes, a question that sounds like a '*Mount Fuji*' question isn't really one at all** and with all the hate we direct towards them we might end up missing it completely. So, what excuse do we give ourselves then, hmmm :)?

What I want to do is present a couple of these '*Non-Mount Fuji*' questions here. While they might seem like one of '*those*' at first, they are actually very reasonable questions, relevant, with concrete solutions. In my opinion, very decent interview fodder as well, but you're likely to fail them just as badly as any 'Mount Fuji' problem if you haven't practiced, so let's have a look.

Here is the first:

> There is a village full of people. One day the priest gathers all the villagers together and declares that their God has told him that some among the villagers are sinners. All sinners will be marked with a sign on their forehead so that all other people will be able to see if a person is a sinner or not, but noone will know if there is a mark on their own forehead. Furthermore, the mark will not be visible in the mirror and one person is forbidden from telling another if they are a sinner. The villagers must work out which among them are sinners at which point all sinners are to leave the village. The village has been given one week and if at the end of the week, there are still sinners present, the whole village will be destroyed.
>
> The villagers go about their own business during the day, however the whole village gathers in the square at the end of every day so that everyone can see if there are still any sinners left among them. After the third such gathering all the sinners pack up their stuff and leave the village.
>
> How many sinners were there? How did you arrive at that number?

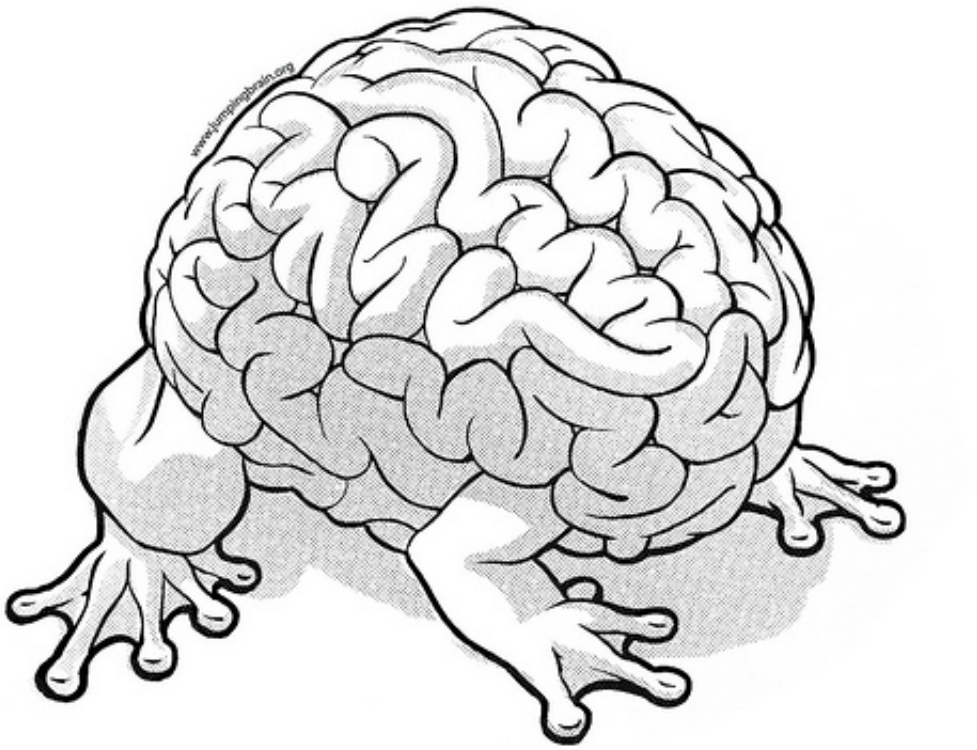
This is a deductive reasoning question. **We tend to use this kind of reasoning all the time when writing software**, so the question is highly relevant to the software development profession. This question has many different forms and is pretty common, so you may have heard it before. If you have you should be able to answer it easily right :)?

Here is the second:

> You are one of several recently arrested prisoners. The warden, a deranged computer scientist, makes the following announcement:
>
> You may meet together today and plan a strategy, but after today you will be in isolated cells and have no communication with one another.
>
> I have setup a "switch room" which contains a light switch, which is either on or off. The switch is not connected to anything.
>
> Every now and then, I will select one prisoner at random to enter the "switch room". This prisoner may throw the switch (from on to off, or vice-versa), or may leave the switch unchanged. Nobody else will ever enter this room.
>
> Each prisoner will visit the "switch room" arbitrarily often. More precisely, for any N, eventually each of you will visit the "switch room" at least N times.
>
> At any time, any of you may declare: "we have all visited the 'switch room' at least once". If the claim is correct, I will set you free. If the claim is incorrect, I will feed all of you to the crocodiles. Choose wisely!
>
> 1. Devise a winning strategy when you know that the initial state of the switch is off.
> 2. Devise a winning strategy when you do not know whether the initial state of the switch is on or off.

This is a distributed coordination problem. The reasoning used to solve this one is relevant when it comes to both distributed computing and concurrency. Infact, I '*stole*' this question from Chapter 1 of the "The Art of Multiprocessor Programming", which is a great book I've been browsing recently (*it really is a very interesting book and I will likely be going through it in a lot more detail, don't worry though, you'll read about it right here when that happens :)*). In my opinion this is an even better question than the last, it feels more '*computery*' for whatever reason. More than that, once you've worked it out, the solution can pretty easily be expressed in code to validate your reasoning (*which is exactly what I did*). Infact, with a little bit of creative thinking, even the first question can be validated through some code. If you can express something in code it's a winner as far as I am concerned.

So What Do We Do With These?



What I want you to do is use this as an opportunity to practice. It doesn't really even matter if you're likely to ever get this kind of question in an interview; these are good questions to give the old programming brain a bit of a workout. Not only is this an opportunity to problem-solve, but it can also be an opportunity to write some quick, interesting code. After going through this exercise, if you ever do see these kinds of questions in an interview, you'll hopefully be able to distinguish them from the '*hated Mount Fuji*' ones :).

So, **get your thinking caps on and try to solve these questions – then post your solutions in the comments below**. The first person to get everything right gets a prize, actually I lie, there is no prize, but apparently incentives don't work anyway, so it doesn't matter. What you will get is a decent mental workout and the satisfaction of solving an interesting problem. Don't worry if you can't get it though, you'll get better with practice as long as you give it a go. Anyway, I won't leave you in the lurch, I'll post my solutions to both of those questions in a few days, including the code I wrote to validate the second one (*I might even write some code for the first one if I get time*). **If you're really brave, then also post how long it took you to work out the answer** (*it doesn't really matter, getting these right at all would be a very decent effort, but if you want to show everyone how much of a genius you really are, this is your chance :)*). I am really looking forward to seeing what you come up with.

Images by Molas and "lapolab"

Related posts:

1. How To Answer A Programming Interview Question And Look Good Doing It
2. An Interview Question That Prints Out Its Own Source Code (In Ruby)
3. Here Is The Main Reason Why You Suck At Interviews

Read more…

On Personal Skills And How Even Riding A Bike Is ... Alan Skorkin May 31, '10, 2:19 PM

It seems that in this industry we're operating under the false assumption that once we learn (*or hear about*) a skill, it is with us for good. You only have to look at a typical developer's resume for an example – it is truly buzzword central. I mean c'mon, just what exactly do you remember about XQuery or iBATIS at this point? I sometimes look at some of my own resumes I have used in the past and can't help being impressed – if only I were really that awesome :). If you're honest with yourself you can quickly classify all those skills into 3 categories:

1. *Stuff you have used recently and know pretty well*
2. *Stuff you used to know pretty well, but haven't used for ages*
3. *Stuff you've had some exposure to, but don't really remember much about*

Only the skills and tech that end up in the first group truly deserve a place in that buzzword list. The problem is that **we don't tend to prune skills/tech off our resumes all that often**. More stuff makes us sound more impressive – who wants to mess with that. This would actually have been ok if we made a conscious decision to include neglected and forgotten skills for marketing purposes, but the sad thing is the vast majority of developers/people truly believe that all those skills deserve a place.

The Power Of The Mind

When we want something to be true, we're really good at convincing ourselves that it actually is. Companies do it all the time, they want to be the market leader so they portray/see themselves as the market leader – the reality of the situation doesn't matter. The same is true for individuals. We have some minor exposure to a skill and we convince ourselves that we're better than a total novice. This may be true initially, but without some reinforcement, this quickly ceases to be the case. That however is irrelevant, we already believe we have some level of expertise and so the buzzword list grows. What's even more interesting are the tricks our mind plays on us when it comes to skills we actually did have some expertise in. You vividly remember when C, Smalltalk or XSLT was your bread and butter, it seems like only yesterday. But, it wasn't yesterday, it was years ago, sometimes many years – **you would be surprised how far those skills can degrade with time**. On some level we're aware of this, after all we're not stupid, but we tell ourselves that it doesn't matter, we're just a little rusty, we can pick it all up again in no time – after all, it's just like riding a bike. Well, let me tell you something about riding bikes.

Just Like Riding A Bike

I learned to ride a bike when I was very young and spent quite a large proportion of my childhood on one. It was simply the thing that we did during summer. I was pretty good (*even if I do say so myself :)*), that's what constant practice will do for you. Then, my family moved to Australia and I didn't ride for many years, until, during one fateful school camp, mountain biking was on the list of activities we had to participate in. I fully expected to ace that activity, my riding memories were still vivid in my mind. **The reality was I sucked rather badly**. Oh, I could ride of course, you never forget the absolute basics, but that was the only thing I really felt confident about. I could no longer judge properly how fast I could take the corners without falling over. I couldn't tell how fast I could safely go downhill, so I took it easy. Riding in a pack with other people seemed scary and unnatural and after only a short while my legs were aching and I was out of breath. Sure, the basics remained, but all the skills, which are built through continued practice and reinforcement, were no longer there. I was essentially reduced to a rank novice.

The Harsh Truth

Every skill is the same, without continued usage and practice they quickly begin to degrade. Your mind begins to page-out the

knowledge and experience to make room for other information. Some goes into "*long term storage*", some is lost completely, but the net result is you degrade towards '*noobhood*'. **The longer you leave a skill untouched, the more you mind will page-out**. It takes longer for skills you were more familiar with and some knowledge is reinforced in other ways so that you never really loose it; but the hard-earned expertise, the things that make you more than just mediocre can disappear with surprising speed. This is both sad and dangerous. It is sad because in software, we devote inordinate amounts of time and effort to acquire and master the skills we use in our work. When the pace of our industry forces us into using wholly different sets of skills/technologies all that time and effort begins to go to waste. Previously useful skills get relegated into disuse and only see the light of day in the resume buzzword section. This is especially sad considering that the "*useless*" old-school skills are often not quite so useless after all. Ruby is all the rage these days, but to make things go fast, what are all the native extensions written in – C, so what's old-school anymore? It is dangerous, because we absolutely refuse to acknowledge that we may not be as expert in some skills as we used to be. Sure **we might loudly proclaim how we have forgotten everything about skill/technology X, but deep down we still believe that we can give these kids a run for their money** when it comes to X (*I am not precisely sure which kids and what money, but you get the picture :)*). This is why we proudly retain X in the buzzword section, but if someone really needed that expertise in a hurry (*like your employer for example*), would you be able to deliver? I am certain you'll be able to get it eventually, but so could any other reasonably competent person, there is a difference between that and expert knowledge.

So What Are We Gonna Do About It

We can simply be scrupulously honest and remove all the skills and tech we're not sure about from the resume. But that actually helps noone, despite what I said above, you with your "*rusty*" skills, are still better than a complete beginner when it comes to those areas. So, by removing that stuff from your resume you're actually underselling yourself – it is wasteful and an easy way out. Rather than doing that, why not make sure your hard-earned skills and knowledge never get quite as rusty again.

As developers, **we tend to do a lot of practice and self-study anyway, but most of us do it in a very diffuse and unfocused fashion**. We learn about the latest and greatest things that strike our fancy and we practice by building stuff that catches our interest. All of that is fine as far as it goes, but we can do a lot better. Here is what I suggest. Engage a in a little self-examination by taking all your skills and sorting them into four buckets:

- **Core skills** – these are the skills you consider primary to the direction you want your career to go. You want to make sure you become increasingly proficient with these and so should devote the most time to practicing them. There will only be a few of these at most (*there is only so much time in a day*).
- **Supporting skills** – these skills support and underpin your core skills. You want to practice these to a lesser extent than the core skills, but you still want to see slow and steady growth. If chosen correctly many of your supporting skills will get a work-out as you practice your core skills, but you should still make sure you devote some time to them exclusively.
- **Peripheral skills** – these are the skills that you consider valuable and worth maintaining. You're not looking to grow

these skills, but you don't really want to lose them either. You want to devote enough time to these so that they don't fall into disuse.
- **Misc** – this is the bucket for "*the rest*". The truth is you can't be an expert in everything, so this should contain the skills that you believe will add the least value to your development going forward.

You essentially end up devoting you practice and study time to your core, supporting and peripheral skills on a rotating basis, dividing your time between all of them based on their importance. It is difficult to sort your skills in this fashion (*and even to work out what constitutes a skill, is building web apps a skill, or should it be specified into building Rails apps or alternatively generalised into being an HTTP protocol expert*), so it might be useful to get some help, but it is an exercise that is eminently worth going through. What it gives you is a set of goals and a strategic framework for where you want to take your skills and your career. Rather than being slave to the latest trends when it comes to your practice, it will allow you to have some focus which can only make your self-study that much more productive (*you'll be able to ignore the fads because you know what you need to focus on and work towards*). The side benefit of this is that after a while, you will be a lot more confident about the buzzword section of you resume and be able to prune it without making it look sparse.

So, which skills should be your core and which should be supporting? This will be different for everyone, I've already told you what I think about fundamentals, so that is something to go on, but ultimately it is up to you to work out what knowledge and skills you want to devote the most time to depending on where you want to take your career. It is perhaps worth exploring this in a lot more detail (*it is fine to have a strategic view, but it doesn't actually tell you what to do :)*) and I might do this in a later post. Of course the other side of the coin is this, even if you do work out what skills you want to focus on, how do you actually go about practicing them, do you just write a lot of code? What if some of the skills I value are not coding related? That aspect of it is what I would call **the tactical view of your growth as a developer – here it is all about deliberate practice** and that is something I will definitely explore further in subsequent posts. Stay tuned.

Images by ItzaFineDay and Pete Prodoehl

Related posts:

1. You Don’t Need Math Skills To Be A Good Developer But You Do Need Them To Be A Great One
2. Building Software Development Expertise – Using The Dreyfus Model
3. On The Mastery Of Teaching