A Spoonful of Python (and Dynamic Programming)

#dynamic programming #fibonacci #memoized recursion #primer #python
2012-01-12

This article was ported from my old Wordpress blog here, If you see any issues with the rendering or layout, please send me an email.

This primer is a third look at Python, and is admittedly selective in which features we investigate (for instance, we don't use classes, as in our second primer on random psychedelic images). We do assume some familiarity with the syntax and basic concepts of the language. For a first primer on Python, see A Dash of Python. We'll investigate some of Python's useful built-in types, including lists, tuples, and dictionaries, and we use them to computing Fibonacci numbers and "optimal" coin change.

Fibonacci Numbers (the Wrong Way)

There's an interesting disconnect between the mathematical descriptions of things and a useful programmatic implementation. Take for instance, the Fibonacci numbers f_n.

$f_1 = f_2 = 1$

$$f_n = f_{n-1} + f_{n-2} \$ for \$ n > 2$$

A direct Python implementation of this definition is essentially useless. Here it is:

```
def fib(n):
   if n <= 2:
      return 1
   else:
      return fib(n-1) + fib(n-2)
```

Recalling our first Python primer, we recognize that this is a very different kind of "for" loop. Indeed, this is a *recursive* loop, which achieves the looping idea by having a function call itself with different arguments. For more examples of this (albeit, in a different language), see our primer A Taste of Racket.

In one mindset that will do us good later in the post, we are simplifying the problem of computing large Fibonacci numbers by breaking it up into a number of sub-problems. The sub-problems require us to compute smaller and smaller Fibonacci numbers, and we build up the sub-problems in some way to arrive at a final solution. This is the essence of dynamic programming, and we'll see a more complicated example shortly.

Unfortunately, nobody in their right mind would use this piece of code. Trying to compute even f_{100}, this algorithm seems to never terminate! Some might throw their hands in the air and claim recursion is totally useless. Indeed, before we get to a working recursive solution (which this author considers more elegant), we will take the indignant view and look for a solution that is totally recursionless.

Listomaina

Python has two built-in types which are list-like. The first is called a *tuple*, and the second is called a *list*. They function almost exactly the same way, but the first is immutable (that means you can't change it after it's created). We covered plain 'ol lists in our first Python primer, but here are a few syntactic reminders, and show off some new list methods.

```
myList = ['a', 2, "gamma", ["wtf another list!", 5]]
myList[-1] = "replacing the list by a string"
myList.append(myList[2:4])
secondList = [2*i for i in range(10)]
```

As in the second line, we can use negative indices to access the elements in reverse order. In the third, we can access *slices* of the list using the colon notation. And in the last line, we have the generally awesome and useful *list comprehensions*. This follows more or less closely with the usual notation for con-

structing lists (of course, using Pythonic syntax). However, one must be careful, because each "for" in a list comprehension literally corresponds to an actual for loop. In other words, the following two bits of code are equivalent:

```
myList = [x*y*z for x in range(100) for y in range(100)
          for z in range(100)]
myList = []
for x in range(100):
   for y in range(100):
     for z in range(100):
        myList.append(x*y*z)
```

On the other hand, tuples have almost precisely the same syntax, except they are constructed with parentheses, and tuple comprehensions have an extra bit of syntax. Here are the same examples as above, except with tuples.

```
myTuple = ('a', 2, "gamma", ("wtf another tuple!", 5))
myTuple[-1] = 2 # not allowed to assign!
myTuple.append(myTuple[2:4]) # not allowed to append!
secondTuple = tuple(2*i for i in range(10))
```

The middle two lines are invalid. Tuples support neither assignment nor appending, and any sort of alteration of a tuple is forbidden. All of the fuss comes from the fact that when you pass a list as a function argument, the function can change the contents of the list. If you're on a large software team and you don't look at what the function does (or use some function whose code you can't view), then you might not not anticipate this. Converting a list to a tuple before passing it is a safety measure.

Indeed, the conversion is in the last line but we could have first create a list, and then converted it to a tuple with "tuple()".

With lists in our pocket, we can come up with a way to compute Fibonacci numbers quickly: we just save the old computations in a list (and do away with all that recursion nonsense).

```
def fib(n):
   fibValues = [0,1]
   for i in range(2,n+1):
      fibValues.append(fibValues[i-1] + fibValues[i-2])
   return fibValues[n]
```

One difference here is that lists index staring at zero, so we have to go up to $n+1$. In fact, some authors set $f_0 = 0$, so this works out just fine. In fact, we can even compute such huge Fibonacci numbers as $f_{100,000}$, and it only takes about four seconds. Compare this to the infinitude of time it took our naive algorithm to compute f_{100} and you'll see why some forsake recursion so quickly.

Before we go on to a more general discussion and a more interesting problem, we note that trick to compute Fibonacci numbers with a single recursive call. We leave this as an exercise to the reader

(hint: instead of the argument being a single number n, pass a tuple containing two useful values). Of course, this method can just as well be done with a loop instead of recursion. The difference is that there is no growing list as we go along, so we say this algorithm takes a constant amount of space (or just *constant space*). And, for completeness, depending on the language certain kinds of recursion do "take up space" in a sense, and Python happens to be a language that does that. For more on this, see tail call optimization.

Dynamic Programming

The feat we just accomplished in computing Fibonacci numbers quickly does generalize to more interesting problems and much *harder* problems. So hard, in fact, that the method has its own name: *dynamic programming*. It's hard to give a precise (and concise) definition for when dynamic programming applies. This author likes to think of it as "the method you need when it's easy to phrase a problem using multiple branches of recursion, but it ends up taking forever since you compute the same old crap way too many times." According to Wikipedia, this means we have "overlapping sub-problems" which are just slightly smaller than the goal. In the case of Fibonacci numbers, this was that we could compute f_n from f_{n-1} and f_{n-2}, but their computations in turn had overlapping sub-computations (both use the value of f_{n-3}).

We avoided the problem of computing stuff multiple times by storing our necessary values in a list, and then building the final solution from the bottom up. This turns out to work for a plethora of problems, and we presently turn to one with some real world applications.

Coin Change

Most readers will have made change for a certain amount of money using a fixed number of coins, and one can ask, "what is the smallest number of coins I can use to make exact change?" The method of picking the largest coins first (and only taking smaller coins when you need to) happens to give the optimal solution in many cases (U.S. coins are one example). However, with an unusual set of coins, this is not always true. For instance, the best way to make change for thirty cents if you have quarters, dimes, and pennies, would be to use three dimes, as opposed to a quarter and five pennies.

This example shows us that the so-called "greedy" solution to this problem doesn't work. So let's try a dynamic algorithm. As with Fibonacci numbers, we need to compute some sub-problems and build off of them to get an optimal solution for the whole problem.

In particular, denote our coin values $v_1, \dots, v_k$ and the amount of change we need to make n. Let us further force $v_1 = 1$, so that there is guaranteed to be a solution. Our sub-problems are to find the minimal number of coins needed to make change if we only use the first i coin values, and we make change for j cents. We will eventually store these values in a two-dimensional array, and call it $\mathrm{minCoins}[i, j]$. In other words, this is a two-dimensional dynamic program, because we have sub-problems that depend on two parameters.

The base cases are easy: how many coins do I need to make change for zero cents? Zero! And how many pennies do I need to make j cents? Exactly j. Now, suppose we know $\mathrm{minCoins}[i-x, j]$ and $\mathrm{minCoins}[i, j-y]$ for all relevant x, y. To determine $\mathrm{minCoins}[i, j]$, we have two choices. First, we can use a coin of value v_i, and add 1 to the array entry $\mathrm{minCoins}[i, j–v_i]$, which contains the best solution if we have $j–v_i$ cents. Second, we can ignore the fact that we have coins of value v_i available, and we can use the solution $\mathrm{minCoins}[i-1, j]$, which gives us the same amount of money either.

In words, the algorithm says we can suppose our last coin is the newly introduced coin, and if that doesn't improve on our best solution yet (above, $\mathrm{minCoins}[i, j-v_i]$) then we don't use the new coin, and stick with our best solution whose last coin is not the new one (above, $\mathrm{minCoins}[i-1, j]$).

Let's implement this idea:

```
def coinChange(centsNeeded, coinValues):
   minCoins = [[0 for j in range(centsNeeded + 1)]
               for i in range(len(coinValues))]
   minCoins[0] = range(centsNeeded + 1)

   for i in range(1,len(coinValues)):
      for j in range(0, centsNeeded + 1):
         if j < coinValues[i]:
            minCoins[i][j] = minCoins[i-1][j]
         else:
            minCoins[i][j] = min(minCoins[i-1][j],
             1 + minCoins[i][j-coinValues[i]])

   return minCoins[-1][-1]
```

The first two lines of the function initialize the array with zeros, and fill the first row with the relevant penny values. The nested for loop runs down each row, updating the coin values as described above. The only difference is that we check to see if the coin value is *too large* to allow us to choose it (j < coinValues[i], or equivalently j – coinValues < 0). In that case, we default to not using that coin, and there is no other way to go. Finally, at the end we return the last entry in the last row of the matrix.

We encourage the reader to run this code on lots of examples, and improve upon it. For instance, we still assume that the first entry in coinValues is 1.

Memoized Recursion

Essentially all of dynamic programming boils down to remembering the solutions to sub-problems. One might ask: do we really need all of this array indexing junk to do that? Isn't there a way that hides all of those details so we can focus on the core algorithm? Of course, the answer to that question is yes, but it requires some more knowledge about Python's built in data types.

In the fill justify problem above, we uniquely identified a sub-problem using two integers, and indexing the rows and columns of a two-dimensional array (a list of lists). However, the fact that it was a two-dimensional array is wholly irrelevant. We could use a one-dimensional list, as long as we had some way to uniquely identify a sub-problem via a key we have access to. The official name for such a look-up table is a *hash table*.

In most standard computer science courses, this is the point at which we might delve off into a long discussion about hash tables (hashing functions, collision strategies, resizing methods, etc.). However for the sake of brevity we recognize that most languages, including Python, have pretty good hash tables built in to them. The details are optimized for our benefit. In Python, they are called *dictionaries*, and they are just as easy to use as lists.

```
dict = {}
dict[1] = 1
dict[2] = 1
dict[3] = 2
dict[(1,2,3,4)] = "weird"
dict["odd"] = 876

if 2 in dict:
   print("2 is a key!")
   print("its value is" + str(dict[2]))

dict2 = {1:1, 2:1, 3:2} # key:value pairs
```

The curly-brace notation is reserved for dictionaries (almost), and it comes with a number of useful functions. As expected, "in" tests for key membership, but there are also a number of useful functions for extracting other kinds of information. We won't need any of them here, however. Finally, the last line shows a quick way of creating new tables where you know the values ahead of time.

We can use a dictionary to save old computations in a function call, replacing the egregious extra computations by lightning fast dictionary look-ups. For our original Fibonacci problem, we can do this as follows:

```
fibTable = {1:1, 2:1}
def fib(n):
   if n <= 2:
      return 1
   if n in fibTable:
      return fibTable[n]
   else:
      fibTable[n] = fib(n-1) + fib(n-2)
      return fibTable[n]
```

We simply check to see if the key in question has already been computed, and if so, we just look the value up in the dictionary. And now we can compute large values of f_n quickly! (Unfortunately, due to

Python's limit on recursion depth, we still can't go as large as the non-recursive solution, but that is a different issue altogether). However we can do better in terms of removing the gunk from our algorithm. Remember, we want to completely hide the fact that we're using dictionaries. This brings us to a very neat Python feature called decorators.

Decorators

A decorator is a way to hide pre- or post-processing to a function. For example, say we had a function which returns some text:

```
def getText():
   return "Here is some text!"
```

And we want to take this function and for whatever text it might return, we want to display it on a webpage, so we need to surround it with the HTML paragraph tags. We might even have a hundred such functions which all return text, so we want a way to get all of them at once without typing the same thing over and over again. This problem is just begging for a decorator.

A decorator accepts a function f as input, and returns a function which potentially does something else, but perhaps uses f in an interesting way. Here is the paragraph example:

```
def paragraphize(inputFunction):
   def newFunction():
      return "<p>" + inputFunction() + "</p>"
   return newFunction

@paragraphize
def getText():
   return "Here is some text!"
```

"paragraphize" is the decorator here, and this function must accept a function as input, and return a function as output. The output function can essentially do whatever it wants, and it will replace the input function. Indeed, the macro "@paragraphize" precedes whatever function we wish to decorate, and it replaces all calls to "getText()" by calls to "newFunction(getText)". In other words, it is shorthand for the following statement

```
getText = paragraphize(getText)
```

Moreover, the function that paragraphize outputs must accept the same number of arguments as the input function accepts. This is to be expected, and the way we signify multiple arguments in Python is with the * notation. Here is an example:

```
def printArgsFirst(f):
   def newFunction(*args):
```

```
      print(args)
      return f(*args)

   return newFunction

@printArgsFirst
def randomFunction(x, y, z):
   return x + y + z
```

The "*args" notation above represents multiple comma-separated values, and when we refer to "args" without the star, it is simply a tuple containing the argument values. If we include the *, as we do in the call to f(*args), it unpacks the values from the tuple and inserts them into the call as if we had called f(x,y,z), as opposed to f((x,y,z)), which calls f with a single argument (a tuple).

Applying this to our memoization problem, we arrive at the following decorator.

```
def memoize(f):
   cache = {}

   def memoizedFunction(*args):
      if args not in cache:
         cache[args] = f(*args)
      return cache[args]

   memoizedFunction.cache = cache
   return memoizedFunction
```

We have a saved cache, and update it whenever we encounter a new call to f(). The second to last line simply allows us to access the cache from other parts of the code by attaching it to memoizedFunction as an attribute (what attributes are requires another discussion about objects, but essentially we're making the dictionary a piece of data that is part of memoizedFunction).

And now we come full circle, back to our original (easy to read) implementation of Fibonacci numbers, with one tiny modification:

```
@memoize
def fib(n):
   if n <= 2:
      return 1
   else:
      return fib(n-1) + fib(n-2)
```

Again, it is unfortunate that we still cannot compute ridiculously huge values of f_n, but one has to admit this is the cleanest way to write the algorithm. We encourage the reader re-solve the fill justify problem using this memoized recursion (and make it easy to read!).

For the adventurous readers, here is a list of interesting problems that are solved via dynamic programming. One common feature that requires it is the justify alignment in word processors (this author has a messy Java implementation of it: 300 lines not including tests! I admit I was a boyish, reckless programmer). All dynamic programs follow the same basic pattern, so with the tools we have now one should be able to go implement them all!

Until next time!

Want to respond? Send me an email, post a webmention, or find me elsewhere on the internet.

DOI: https://doi.org/10.59350/wfw2f-3x627

← Previous: Numerical Integration

Next: Word Segmentation, or Makingsenseofthis →