

Natasha Eilbert
Algorithms: Design and Practice
Deepak Kumar
April 21, 2008

Memoization: Speed It Up

Speed is an ever-present concern in computing. Everyone wants programs and software – and of course the internet – to be faster. A technique called *memoization* can increase the efficiency of a particular class of programs. The name was thought up by Donald Michie in 1968.¹ Shortly thereafter, B.A. Sheil started applying it to language parsing.² Memoization works by replacing repeated function calls within a given program with a lookup table for these function calls' return values. The analogy of the lookup table is that of a notepad full of – guess what – *memos*.³ A memoized function first glances through its notepad memos – i.e., a lookup table – to see if it has ever been called before with the same arguments. If so, it simply returns the already scribbled-down return value. Otherwise, it goes through with the calculation of the function, adding this new value to its list of memos.

By locating already calculated return values in a lookup instead of re-calculating them by traversing the function again, the memoization process can significantly reduce a program's overall time. The technique is beneficial mainly in programs that require repeat function calls handling the same parameters, such as recursively defined functions, and is especially helpful for functions requiring many operations and calculations.⁴ While there are some situations for which memoization does not improve efficiency, there are others where, due to memoization's limitations, it is actually inappropriate to use as it will disrupt a program's intended functioning.

Memoization should only be used for what are called *referentially transparent* functions, ones that have no effects save that of producing a return value.⁵ For instance, they can not change variables used elsewhere in the program nor output a message to the screen. The functions must also produce the same result for a given set of parameters. It makes sense that using a lookup table's value for a function instead of executing the function itself would not be appropriate if the function does work besides returning a value or does not always return the same value.

There is one additional consideration when using memoization. The fact is, it doesn't come for free; there is a cost associated with the significant speed-up of a program through memoization. Namely, data must be stored in the lookup table – or in our analogy, in the tons of memos that would become scattered around the house – so a

¹ "Memoization." 13 Apr 2008. Wikipedia. 5 Apr 2008. <<http://en.wikipedia.org/wiki/Memoization>>.

² Evidenced, in 1976, Sheil, B.A. published "Observations on context-free parsing." (Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University.)

³ "Memoization." 13 Apr 2008. Algorithmist. 2 Sept 2007.
<<http://www.algorithmist.com/index.php/Memoization>>.

⁴ Moore, Paul. "Memoizing (cacheing) function return values." Active State Programmer Network. 13 Apr 2008. Active State Software. 16 Oct 2001.
<<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52201>>.

⁵ "Memoization." 13 Apr 2008. Wikipedia. 5 Apr 2008. <<http://en.wikipedia.org/wiki/Memoization>>.

memoized function requires more memory than a non-memoized one. In exchange for improvements in time, there are drawbacks in space.⁶

One might ask why, instead of memoizing a program, we do not simply generate and fill the entire lookup table at the beginning of the program, and then use it instead of calculating the functions. This is a great waste of time and space. When memoizing, we calculate and store only the values we need – and if we happen to need them again, then they are available.⁷

Let's look at a concrete example of memoization in use. We can write a function, `factorial(n)` that calculates the factorial of a whole number `n` in pseudocode as below.

```
define factorial(n):  
  if n is 0, then return 1          // since 0! = 1  
  otherwise, return n * factorial(n-1)
```

For the basis case, the factorial of zero is, as it is defined to be, the value one and the factorial of any other number `n` is the result of multiplying `n` and the factorial of the previous number, `n – 1`.

Now this function is fine and dandy and works correctly. However, let's say we wanted to find the factorial of numbers three and four (we could have chosen much larger numbers where difference in efficiency is more drastic, but for simplicity stuck with small ones). When we call `factorial(3)`, the computer must do the work of calculating the factorial for 3, 2, 1 and 0. Similarly, when we call `factorial(4)`, the computer must calculate the factorial for 4, 3, 2, 1 and 0 – even though just seconds ago we calculated the factorial for 3, 2, 1, and 0. In calling `factorial(4)` we're doing a lot of repeat work. Note that even if we call the function on a smaller number, we will still be doing extra work; had we called `factorial(2)`, we would have to calculate the factorials for 2, 1, and 0, which have already done. How can we avoid doing all this excess work? Let's first take a look at exactly how much extra work we're doing in the particular case of our recursive factorial function.

Generalizing the above function calls of factorial two, three and four, we see that at any function call for `factorial(n)`, the computer must actually call the factorial function `n+1` times. Furthermore, each of the `n+1` times that the function is called, the computer must expend energy both on function overhead when creating the function and on all calculations and flow of control statements within the function (e.g. comparing `n` to 0 in the if-statement or calculating `n – 1`).⁸

This recursive factorial function seems a perfect candidate for reconstructing using memoization. We have already shown that this recursive function is time-consuming in any case and that as it stands, this puffy, middle-aged, heart-attack prone function is doing altogether too much work repeating tasks it's already done. Let's help it out. At the same time, we'll be helping ourselves by increasing the speed of our factorial-calculating tool.

⁶ "Memoization." 13 Apr 2008. Wikipedia. 5 Apr 2008. <<http://en.wikipedia.org/wiki/Memoization>>.

⁷ Neuburg, Matt. "Memoizing: How to Take Good Notes." 13 Apr 2008. <<http://www.tidbits.com/matt/rbd/memoizing.html>>.

⁸ "Memoization." 13 Apr 2008. Wikipedia. 5 Apr 2008. <<http://en.wikipedia.org/wiki/Memoization>>.

Before going any further, we should check that our function is within limitations of memoization – in other words, we should make sure it is referentially transparent. Does it produce any effects other than its return value? No. Does it always return the same value for a given input n ? In my experience, $\text{factorial}(x) = \text{factorial}(x)$ for any valid x , and likewise in this pseudocode, so yes. Then good, we can use memoization here to help out our huffy red-cheeked friend, and ourselves.

Below is the memoized edition of our factorial function. For now, think of `lookupTable` as some data structure in which already calculated values have been stored and newly calculated values will be stored, based on corresponding function parameters.

```
define factorial(n):
  if n is in lookupTable, then return lookupTable[n]
  otherwise if n is 0, then return 1
  otherwise:
    result = n * factorial(n-1)
    lookupTable[n] = result #store result in lookupTable with reference n
  return result
```

To find the factorial of a number n , the function first checks its memo pad, `lookupTable`, to see if it has already calculated the specified factorial. If it has, it simply returns the value located in the `lookupTable`. Otherwise, it calculates the value and, when not in the basis case, stores the value in the `lookupTable` for later retrieval and then returns it. Since it is faster to return an explicit value than look it up in a table, the base case returns the value directly.

Earlier, we said the `lookupTable` should be some type of container to hold our results. What kind of structure would be best? Since we would like to look up return values based on the arguments given to the function, we can use these arguments as keys in an associative list or dictionary available in our language of choice.

Notice that in the particular case of the factorial, since a factorial for any number inherently relies on the factorial of the previous number, we end up adding values to the `lookupTable` only sequentially. So we might think to simply use a list structure for the `lookupTable`, using the input parameter (with offset one) as the index. However, some functions other than factorial may not add parameters in order, and having to calculate the index based on a complex algorithm is unnecessary. Further, there may be multiple input arguments, which are not necessarily integers. Hence a dictionary lookup makes memoizing more applicable to many functions, no matter the kind or number of their input parameters.

In the above example, we have indeed memoized our function. Yet we have spent time and energy hand-memoizing it. To expedite the process on the programming end of memoizing, we can make a general function that will memoize any function, shown below. Here, `Function` is both a function and an object, so it can have the necessary `lookupTable` associated with it.

```
define memoized_function_call(Function, args):
  if Function.lookupTable does not exist:
    add lookupTable for Function
```

```

if Function.lookupTable[args] does not exist:
    // calculate and store resulting return value with associated arguments
    Function.lookupTable[args] = Function(args)
return Function.lookupTable[args]9

```

This function gives you the basic idea of automated memoization of a function. It works much like our hand-memoized factorial, but is generalized for use with any function. Specifically, we take in a function with specified arguments. If the function-object does not yet have a lookupTable, we provide one. If the function's lookupTable already contains values for the provided arguments, we return those values. Otherwise, we calculate the results using our function, and then store and return the results. Essentially, the memoized_function_call follows the concept of memoization: it always tries to use the function-objects' lookupTable to find results, but if an entry is not yet available it calculates the results directly, making sure to store them in the lookupTable first.

For the pseudocode shown here, at any point in the program, to call the memoized function we must explicitly write memoized_function_call(theFunction, theArgs). The details of implementation will depend on the programming language of choice. In some languages, such as Python, this can be done more neatly by using classes and/or function definitions within definitions, to avoid having to change existing functions. However, the above pseudocode illustrates the essentials of automated memoization. For more information on the Python implementation, see the Appendix.

Figure 1 shows timing data for a Python implementation of memoization. I timed 10,000 iterations of factorial(300) for various functions. I used a dictionary data structure as my lookupTable. For each function, I show data where I clear the dictionary and where I do not. So for the runs in which I do not clear the data, all the calculation work is done by the function itself. In the versions where I do clear the data, if the function is memoized (whether by hand, as in mem_factorial, or by automation, as in Mem_class_factorial) then all but the first run simply use the lookup table instead of calculating the result using the function. In real-life usage, the distribution of work would likely be between these two extremes, with some work done by the function itself and some by the lookup table.

Function	Dictionary Cleared	Time (sec)
harness	Yes	0.01
harness	No	0
factorial	Yes	3.635
factorial	No	3.626
mem_factorial	Yes	3.635
mem_factorial	No	0.02
Mem_class_factorial	Yes	10.636
Mem_class_factorial	No	0.02

Figure 1. Timing data for memoized Python factorial functions and harness. Data shows 10,000 iterations of factorial(300).

⁹ "Memoization." 13 Apr 2008. Wikipedia. 5 Apr 2008. <<http://en.wikipedia.org/wiki/Memoization>>.

The results of this lookup are generally in accord with the concepts of memoization. The harness is present to show the overhead of a function call, which appears to be minimal. The factorial function does not actually use a dictionary, so with or without clearing the dictionary it takes about 3.6 seconds. The hand-memoized function `mem_factorial`, when clearing the dictionary, takes about the same time. However, once we start using solely the lookup table (for all but the first run) and not having to deal with operations, the time is dramatically reduced to only about 0.02 seconds. The automatically-memoized function `Mem_class_factorial`, when clearing the dictionary, seems to take longer when executing the actual function, perhaps because it must enter into an object and not just a function. However, once we stop clearing the dictionary, the speed is drastically increased to 0.02 seconds. Hence memoization seems to really be working here to speed up the program.

Fortunately, there are a number of uses for memoization outside of just speeding up factorial calculations. There is an alternate version of memoization, called continuous memoization. Instead of using the `lookupTable` only when a function's arguments exactly match those present in the table, this version will accept arguments that are close to some values already in the table.¹⁰ It will look up and then approximate a return value based on similar arguments present in the `lookupTable`. This method is more efficient but less accurate than normal memoization.

Memoization can also be used as a simple way of profiling a program. There are a number of advanced ways to profile a program, or analyze the components taking the most time in a program. However, there is a simple, fast way to do program analysis using memoization. To evaluate the necessity of a given function we create two versions of the program, the original version of the program with the original function, and a new version of the program in which the function has been memoized. We then time the original function, fill the function's lookup table by running the memoized version a number of times, and then time the memoized version of the program. For functions where memoization is applicable, this test estimates the speed of the program with the function not present. If the program is not significantly faster using memoization, then this function has little room for improvement in terms of efficiency, and we can go on to improve other functions with more potential time savings.¹¹

Another application is in natural language processing, where memoization can speed up the computer's ability to parse grammar trees, or find grammatical matches to written sentences. Parsing works by enumerating a list of grammatical rules, such as "Noun Phrase → Adjective Noun", which says that a noun phrase can consist of an adjective followed by a noun. In order to match a sentence to such rules, one must match a word to, in this example, a part of speech term. In analyzing a number of rules, a program may have to match a phrase with the same rule a number of times over again. Instead of always using a function to match the rule, a program can create a lookup table

¹⁰ "Continuous Memoization." 13 Apr 2008. Mitsubishi Electric Research Laboratories. 30 Oct 2002. <<http://www.merl.com/projects/memoization>>.

¹¹ "Bricolage: Memoization." 13 Apr 2008. The Perl Journal. 1999. <<http://perl.plover.com/MiniMemoize/memoize.html>>.

of past matches.¹² By using this lookup table memoization, language parsing can be sped up.

In sum, memoization is a useful tool for improving the efficiency of a number of different programming applications. The concept is relatively recent, and it continues to be useable and to have new applications, especially in the context of language parse trees. We should remember, though, that its effects are limited to certain types of functions, as well as that it represents a trade-off between time and space efficiency.

¹² Johnson, Mark. "Memoization of Top-down Parsing." Computational Linguistics V. 21.3. 25 Apr 1995. Also "Memoization." 13 Apr 2008. Wikipedia. 5 Apr 2008. <<http://en.wikipedia.org/wiki/Memoization>>.

Works Cited

“Bricolage: Memoization.” 13 Apr 2008. The Perl Journal. 1999.
<<http://perl.plover.com/MiniMemoize/memoize.html>>.

“Continuous Memoization.” 13 Apr 2008. Mitsubishi Electric Research Laboratories. 30 Oct 2002. <<http://www.merl.com/projects/memoization>>.

Johnson, Mark. “Memoization of Top-down Parsing.” *Computational Linguistics* V. 21.3. 25 Apr 1995.

“Memoization.” 13 Apr 2008. Algorithmist. 2 Sept 2007.
<<http://www.algorithmist.com/index.php/Memoization>>.

“Memoization.” 13 Apr 2008. Wikipedia. 5 Apr 2008.
<<http://en.wikipedia.org/wiki/Memoization>>.

Moore, Paul. “Memoizing (cacheing) function return values.” Active State Programmer Network. 13 Apr 2008. Active State Software. 16 Oct 2001.
<<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52201>>.

Neuburg, Matt. “Memoizing: How to Take Good Notes.” 13 Apr 2008.
<<http://www.tidbits.com/matt/rbd/memoizing.html>>.

Appendix

Below is the Python program I used to time the memoized and non-memoized factorial functions.

```
# memoize.py
# Computes factorial of a number in a straightforward way,
# using hand-written memoization, and using automated memoization.
```

```
from time import time
```

```
fact_lookup = { }
```

```
def timeFunc(func, num, arg, clear, funcIsClass = False):
    midTimes = 0
    start = time()

    # Test the function num number of times
    for i in range(num):
        func(arg)

        # Clear dictionary if "clear" selected,
        # making sure to subtract the time this takes
        startMid = time()
        if clear:
            if func == mem_factorial:
                fact_lookup.clear()
            elif func == factorial: #class only
                if funcIsClass:
                    func.lookup.clear()
            else:
                fact_lookup.clear() #so harness clears a dict too
        endMid = time() - startMid
        midTimes += endMid

    # Calculate the time taken
    elaps = time() - start - midTimes
    return elaps
```

```
def timeFuncs(span, args = (0, 400, 800)):
    """
    Times each function (for all specified arguments
    and for clear-dictionary flag set and not set).
```

```

span    number of times to execute
args    tuple of arguments for function
"""

# For each function, argument, and clear-flag...
for func in (factorial, mem_factorial, harness):
    for arg in args:
        for clear in (True, False):

            # Time the function
            fact_time = timeFunc(func, span, arg, clear)

            # Clear dictionary between trials
            fact_lookup.clear()

            # Print out results
            #print "%10s -- %3d, %5s ... %8.4f" %(
            print "%10s\t%3d\t%5s\t%8.4f" %(
                str(func).split()[1][:10], arg, clear, fact_time)

def timeClass(span, func, args = (0, 150, 300)):
    """Timing mechanism like timeFuncs, but tailored to
    time a function-object.

    Use like:
    factorial = Memoize(factorial)
    timeClass(300, factorial)"""

    for arg in args:
        for clear in (True, False):
            fact_time = timeFunc(func, span, arg, clear, True)
            func.lookup.clear()
            #print "%10s -- %3d, %5s ... %8.4f" %(
            print "%10s\t%3d\t%5s\t%8.4f" %(
                str(func).split()[1][:10], arg, clear, fact_time)

def harnessWithDict(arg):
    """To see how much time clearing dictionary takes.
    There is additional time caused by adding items to dict."""

```

```

    for i in range(arg):
        fact_lookup[arg] = arg

def harness(arg):
    pass

#####
##
## Factorial functions -- non-memoized, hand-memoized, and
## automatically memoized.
##
#####

def factorial(n):
    """Straightforward approach to computing a factorial.
    Works correctly, but is inefficient."""
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

def mem_factorial(n):
    """Computes factorial using explicit memoization.
    Works fast, but is not easily generalizable."""

    # Note that if-else structure is done for clarity, not necessity
    if n in fact_lookup: # n is a key in lookup dictionary
        return fact_lookup[n]
    elif n == 0:
        return 1
    else:
        result = n * factorial(n-1)
        fact_lookup[n] = result # store result in lookup based on input n
    return result

class Memoize:
    def __init__(self, fn):
        self.lookup = {}
        self.fn = fn
    def __call__(self, *params):
        if params in self.lookup:
            return self.lookup[params]
        else:
            #print self.fn

```

```
    value = self.fn(*params) # calculate new value
    self.lookup[params] = value # store results in lookup
return value
```

class Func:

```
    """Class having a function and a lookup table.
```

```
    For use with memoizing."""
```

```
    def __init__(self, fn):
```

```
        self.lookup = {}
```

```
        self.fn = fn
```

def memoizeFunc(Fn):

```
    """Creates a function that will behave as original function,
    but with memoizing capabilities. Then returns the modified
    function.
```

```
    Fn    a Func object
```

```
    Use like theFunction = memoizeFunc( Func(theFunction) ).
```

```
    Must use the original function's exact name."""
```

```
    def newfn(args):
```

```
        if not Fn.lookup: #not exist
```

```
            Fn.lookup = {} #not nec
```

```
        try:
```

```
            return Fn.lookup[args]
```

```
        except KeyError:
```

```
            val = Fn.fn(args)
```

```
            Fn.lookup[args] = val
```

```
            return val
```

```
    return newfn
```

def tester1(times, args):

```
    """Tests functions using timeFuncs."""
```

```
    for i in times:
```

```
        print "Iterating %d times:" %i
```

```
        timeFuncs(i, args)
```

```
        fact_lookup.clear()
```

```
        print
```

def tester2(times, args):

```
    """Tests function-object using timeClass."""
```

```
    for i in times:
```

```
        print "Iterating %d times:" %i
```

```
        timeClass(i, factorial, args)
```

```
        factorial.lookup.clear()
```

```
        print
```

```
if __name__ == '__main__':  
    times = range(0, 10001, 5000)  
    args = (0, 150, 300)  
    tester1(times, args)  
    factorial = Memoize(factorial)  
    tester2(times, args)
```