Algorithms/Dynamic Programming

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A

Dynamic programming can be thought of as an optimization technique for particular classes of backtracking algorithms where subproblems are repeatedly solved. Note that the term *dynamic* in dynamic programming should not be confused with dynamic programming languages, like Scheme or Lisp. Nor should the term *programming* be confused with the act of writing computer programs. In the context of algorithms, dynamic programming always refers to the technique of filling in a table with values computed from other table values. (It's dynamic because the values in the table are filled in by the algorithm based on other values of the table, and it's programming in the sense of setting things in a table, like how television programming is concerned with when to broadcast what shows.)

Fibonacci Numbers

Before presenting the dynamic programming technique, it will be useful to first show a related technique, called **memoization**, on a toy example: The Fibonacci numbers. What we want is a routine to compute the nth Fibonacci number:

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
```

By definition, the nth Fibonacci number, denoted is

How would one create a good algorithm for finding the nth Fibonacci-number? Let's begin with the naive algorithm, which codes the mathematical definition:

```
// fib -- compute Fibonacci(n)
function fib(integer n): integer
  assert (n >= 0)
  if n == 0: return 0 fi
  if n == 1: return 1 fi

  return fib(n - 1) + fib(n - 2)
end
```

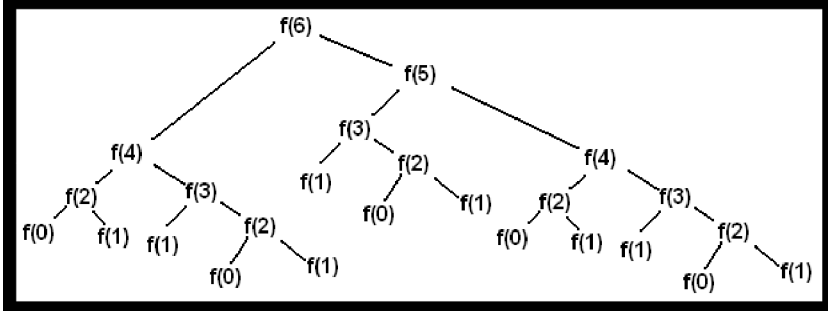
This code sample is also available in Ada.

Note that this is a toy example because there is already a mathematically closed form for :

where:

This latter equation is known as the Golden Ratio. Thus, a program could efficiently calculate for even very large n. However, it's instructive to understand what's so inefficient about the current algorithm.

To analyze the running time of `fib` we should look at a call tree for something even as small as the sixth Fibonacci number:



Every leaf of the call tree has the value 0 or 1, and the sum of these values is the final result. So, for any n, the number of leaves in the call tree is actually itself! The closed form thus tells us that the number of leaves in **`fib`**(n) is approximately equal to

(Note the algebraic manipulation used above to make the base of the exponent the number 2.) This means that there are far too many leaves, particularly considering the repeated patterns found in the call tree above.

One optimization we can make is to save a result in a table once it's already been computed, so that the same result needs to be computed only once. The optimization process is called memoization and conforms to the following methodology:

Memoization Methodology

1. Start with a backtracking algorithm
2. Look up the problem in a table; if there's a valid entry for it, return that value
3. Otherwise, compute the problem recursively, and then store the result in the table before returning the value

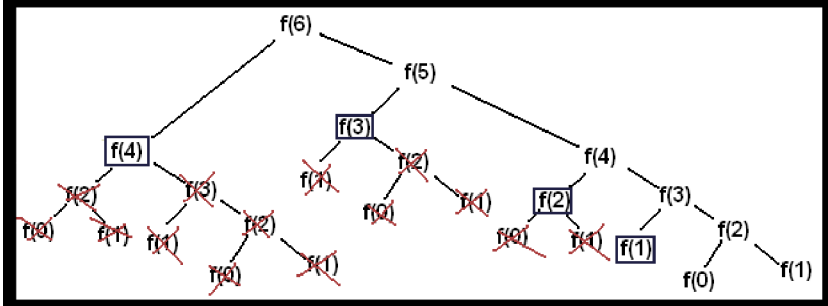
Consider the solution presented in the backtracking chapter for the Longest Common Subsequence problem. In the execution of that algorithm, many common subproblems were computed repeatedly. As an optimization, we can compute these subproblems once and then store the result to read back later. A recursive memoization algorithm can be turned "bottom-up" into an iterative algorithm that fills in a table of solutions to subproblems. Some of the subproblems solved might not be needed by the end result (and that is where dynamic programming differs from memoization), but dynamic programming can be very efficient because the iterative version can better use the cache and have less call overhead. Asymptotically, dynamic programming and memoization have the same complexity.

So how would a fibonacci program using memoization work? Consider the following program ($f[n]$ contains the nth Fibonacci-number if has been calculated, -1 otherwise):

```
function fib(integer n): integer
  if n == 0 or n == 1:
    return n
  else-if f[n] != -1:
    return f[n]
  else
    f[n] = fib(n - 1) + fib(n - 2)
    return f[n]
  fi
end
```

This code sample is also available in Ada.

The code should be pretty obvious. If the value of fib(n) already has been calculated it's stored in f[n] and then returned instead of calculating it again. That means all the copies of the sub-call trees are removed from the calculation.



The values in the blue boxes are values that already have been calculated and the calls can thus be skipped. It is thus a lot faster than the straight-forward recursive algorithm. Since every value less than n is calculated once, and only once, the first time you execute it, the asymptotic running time is . Any other calls to it will take since the values have been precalculated (assuming each subsequent call's argument is less than n).

The algorithm does consume a lot of memory. When we calculate fib(n), the values fib(0) to fib(n) are stored in main memory. Can this be improved? Yes it can, although the running time of subsequent calls are obviously lost since the values aren't stored. Since the value of fib(n) only depends on fib(n-1) and fib(n-2) we can discard the other values by going bottom-up. If we want to calculate fib(n), we first calculate fib(2) = fib(0) + fib(1). Then we can calculate fib(3) by adding fib(1) and fib(2). After that, fib(0) and fib(1) can be discarded, since we don't need them to calculate any more values. From fib(2) and fib(3) we calculate fib(4) and discard fib(2), then we calculate fib(5) and discard fib(3), etc. etc. The code goes something like this:

```
function fib(integer n): integer
  if n == 0 or n == 1:
    return n
  fi

  let u := 0
  let v := 1

  for i := 2 to n:
    let t := u + v
    u := v
    v := t
  repeat

  return v
end
```

This code sample is also available in Ada.

We can modify the code to store the values in an array for subsequent calls, but the point is that we don't *have* to. This method is typical for dynamic programming. First we identify what subproblems need to be solved in order to solve the entire problem, and then we calculate the values bottom-up using an iterative process.

Longest Common Subsequence (DP version)

The problem of Longest Common Subsequence (LCS) involves comparing two given sequences of characters, to find the longest subsequence common to both the sequences.

Note that 'subsequence' is not 'substring' - the characters appearing in the subsequence need not be consecutive in either of the sequences; however, the individual characters do need to be in same order as appearing in both sequences.

Given two sequences, namely,

$$X = \{x_1, x_2, x_3, ..., x_m\} \text{ and } Y = \{y_1, y_2, y_3, ..., y_n\}$$

we define:

$$Z = \{z_1, z_2, z_3, ..., z_k\}$$

as a subsequence of X, if all the characters $z_1, z_2, z_3, ..., z_k$, appear in X, and they appear in a strictly increasing sequence; i.e. z_1 appears in X before z_2, which int turn appears before z_3, and so on. Once again, it is not necessary for all the characters $z_1, z_2, z_3, ..., z_k$ to be consecutive; they must only appear in the same order in X as they are in Z. And thus, we can define $Z = \{z_1, z_2, z_3, ..., z_k\}$ as a common subsequnce of X and Y, if Z appears as a subsequence in both X and Y.

The backtracking solution of LCS involves enumerating all possible subsequences of X, and check each subsequence to see whether it is also a subsequence of Y, keeping track of the logest subsequence we find [see *Longest Common Subsequence (exhaustive version)*]. Since X has m characters in it, this leads to 2^m possible combiations. This approach, thus, takes exponential time and is impractical for long sequences.

Matrix Chain Multiplication

Suppose that you need to multiply a series of matrices together to form a product matrix :

This will require multiplications, but what is the fastest way we can form this product? Matrix multiplication is associative, that is,

for any , and so we have some choice in what multiplication we perform first. (Note that matrix multiplication is *not* commutative, that is, it does not hold in general that .)

Because you can only multiply two matrices at a time the product can be paranthesized in these ways:

Two matrices and can be multiplied if the number of columns in equals the number of rows in . The number of rows in their product will equal the number rows in and the number of columns will equal the number of columns in . That is, if the dimensions of is and has dimensions their product will have dimensions .

To multiply two matrices with each other we use a function called matrix-multiply that takes two matrices and returns their product. We will leave implementation of this function alone for the moment as it is not the focus of this chapter (how to multiply two matrices in the fastest way has been under intensive study for several years [TODO: propose this topic for the *Advanced* book]). The time this function takes to multiply two matrices of size and is proportional to the number of scalar multiplications, which is proportional to . Thus, paranthezation matters: Say that we have three matrices , and . has dimensions , has dimensions and has dimensions . Let's paranthezise them in the two possible ways and see which way requires the least amount of multiplications. The two ways are

, and
.

To form the product in the first way requires 75000 scalar multiplications (5*100*100=50000 to form product and another 5*100*50=25000 for the last multiplications.) This might seem like a lot, but in comparison to the 525000 scalar multiplications required by the second parenthesization (50*100*100=500000 plus 5*50*100=25000) it is miniscule! You can see why determining the parenthesization is important: imagine what would happen if we needed to multiply 50 matrices!

Forming a Recursive Solution

Note that we concentrate on finding a how many scalar multiplications are needed instead of the actual order. This is because once we have found a working algorithm to find the amount it is trivial to create an algorithm for the actual parenthesization. It will, however, be discussed in the end.

So how would an algorithm for the optimum parenthesization look? By the chapter title you might expect that a dynamic programming method is in order (not to give the answer away or anything). So how would a dynamic programming method work? Because dynamic programming algorithms are based on optimal substructure, what would the optimal substructure in this problem be?

Suppose that the optimal way to parenthesize

splits the product at :

.

Then the optimal solution contains the optimal solutions to the two subproblems

That is, just in accordance with the fundamental principle of dynamic programming, the solution to the problem depends on the solution of smaller sub-problems.

Let's say that it takes scalar multiplications to multiply matrices and , and is the number of scalar multiplications to be performed in an optimal parenthesization of the matrices . The definition of is the first step toward a solution.

When , the formulation is trivial; it is just . But what is it when the distance is larger? Using the observation above, we can derive a formulation. Suppose an optimal solution to the problem divides the matrices at matrices k and k+1 (i.e.) then the number of scalar multiplications are.

That is, the amount of time to form the first product, the amount of time it takes to form the second product, and the amount of time it takes to multiply them together. But what is this optimal value k? The answer is, of course, the value that makes the above formula assume its minimum value. We can thus form the complete definition for the function:

A straight-forward recursive solution to this would look something like this *(the language is)*:

```
function f(m, n) {

    if m == n
        return 0

    let minCost :=

    for k := m to n - 1 {
        v := f(m, k) + f(k + 1, n) + c(k)
        if v < minCost
            minCost := v
    }
    return minCost
}
```

This rather simple solution is, unfortunately, not a very good one. It spends mountains of time recomputing data and its running time is exponential.

Using the same adaptation as above we get:

```
function f(m, n) {

    if m == n
        return 0

    else-if f[m,n] != -1:
      return f[m,n]
    fi

    let minCost :=

    for k := m to n - 1 {
        v := f(m, k) + f(k + 1, n) + c(k)
        if v < minCost
            minCost := v
    }
    f[m,n]=minCost
    return minCost
}
```

Parsing Any Context-Free Grammar

Note that special types of context-free grammars can be parsed much more efficiently than this technique, but in terms of generality, the DP method is the only way to go.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A