

Dynamic Programming

Martijn van Steenbergen

November 10, 2005

Abstract

The general concepts behind dynamic programming are shown by discussing several implementations of the Fibonacci sequence. Next, a problem from information theory known as the Levenshtein distance is discussed in detail. Finally, an overview of other dynamic programming-related problems are given.

Contents

1	Introduction	3
2	The Fibonacci sequence	3
2.1	Divide and conquer	4
2.2	Dynamic programming	5
2.3	Memoization	6
2.4	A constant-space algorithm	8
3	Ingredients of dynamic programming	9
4	Levenshtein distance	10
5	Other examples	12
6	Conclusion	12

1 Introduction

The name ‘dynamic programming’ is somewhat misleading. One may think that it is a certain way of programming, but it’s actually a specific strategy to solve certain algorithmic problems. You can point at an algorithm and say, ‘that algorithm uses dynamic programming’. This article discusses what dynamic programming is exactly, its different interpretations, and some common practical examples.

Throughout this article, snippets of code will be presented to illuminate the examples given. The snippets will be in Java, so they can be easily understood and directly implemented by the reader.

2 The Fibonacci sequence

To demonstrate dynamic programming and related strategies, we will discuss several implementations of the Fibonacci sequence. Named after Fibonacci, an Italian mathematician whose real name was Leonardo Pisano, this is a well-known sequence of numbers related to the golden ratio with many wondrous properties. [Wei99] Fibonacci wrote about this sequence in his book *Liber Abaci*, published in 1202. It is defined recursively as follows:

$$\begin{aligned}F_1 &= 1 \\F_2 &= 1 \\F_n &= F_{n-1} + F_{n-2}\end{aligned}$$

Following this definition, the first ten elements of the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34 and 55.

We are going to look at several implementations of this recursively defined sequence to demonstrate the several approaches. These examples are used for demonstration purposes only. There are much more efficient implementations. [Wel86], p. 62 for example gives a direct formula for calculating F_n :

$$F_n = \text{round}\left(\frac{\phi^n}{\sqrt{5}}\right) \text{ where } \phi \text{ is the golden ratio } \frac{1}{2}(1 + \sqrt{5}).$$

This implementation probably uses constant time and space, depending on the implementation of `round()` and the power function, as opposed to the linear and exponential implementations in the following sections.

Sometimes you will find the Fibonacci sequence defined to start with 0, 1 instead of 1, 1. Also, you can extend the sequence into the negative numbers. [Wei99] However, to keep the examples easier to understand, only positive Fibonacci numbers are used, and the indexing of the Fibonacci numbers

always starts at 1.

2.1 Divide and conquer

Divide and conquer (D&C), like dynamic programming, is a strategy to solve certain problems. With this approach, you recognise that the solution to the problem you have to solve can be expressed as a function of solutions to smaller problems. [CLRS01] For the Fibonacci sequence, this is trivial, because the elements of the sequence are defined this way. A direct D&C implementation for the Fibonacci sequence is:

```
int fib(int n) {
    if (n == 1 || n == 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

However, the running time of this implementation explodes for large n . You can easily see that it calculates certain Fibonacci numbers several times. Figure 1 shows the call tree for `fib(6)`.

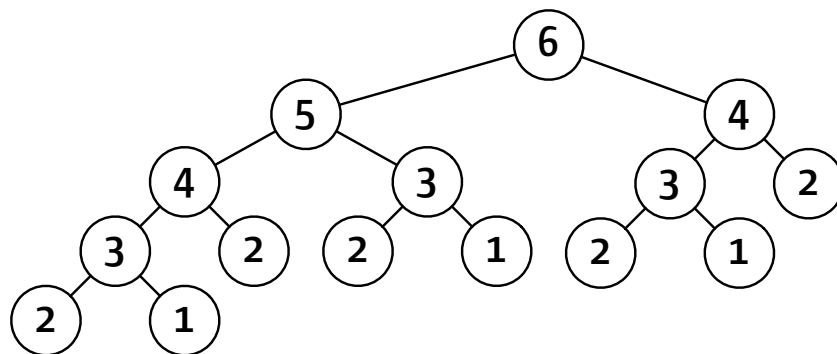


Figure 1: The method call tree for the divide-and-conquer approach to calculating Fibonacci numbers. Every node represents a method call. The number in the node is the argument with which the method was called.

`fib(2)`, for example, gets called five times, even though the answer to `fib(2)` is always the same, so there's no need to recalculate it. How many times does `fib` get called in total? Instead of counting the number of nodes in the tree, we can figure it out for the more general `fib(n)`. If you take another look at the code, you can see that `fib` either returns 1, or the sum of two other values. This means that the total answer must be composed of F_n 1's and $F_n - 1$ additions. This is exactly what the tree shows: the leaves correspond to the 1's, and the internal nodes correspond to the additions.

Indeed, there are $F_6 = 8$ leaves and $F_6 - 1 = 7$ internal nodes.

Recall the direct formula for F_n and note that $F_n \in O(\phi^n)$. This means that this approach's running time is exponential. Its space usage is $O(n)$, because the tree is a call tree, so the maximum space this algorithm needs corresponds to the depth of the tree, which is exactly $n - 1$.

2.2 Dynamic programming

This is where dynamic programming (DP) comes in. By applying DP, you realise that there are *overlapping subproblems* and build your algorithm in such a way that these subproblems—although needed several times—are calculated just once. How can we take advantage of this knowledge and reduce the running time of our previous implementation? One option is to create an array `a` of length n , where `a[i]` contains F_{i+1} . Note that the indices for F are 1-relative, while the indices for `a` are 0-relative. We then put the two 'atomic' elements of the sequence in the array first and fill the rest of the array bottom-up:

```
int fib(int n) {
    // Check for atomic cases
    if (n == 1 || n == 2)
        return 1;

    int[] a = new int[n];
    a[0] = 1;
    a[1] = 1;
    for (int i = 2; i < n; i++)
        a[i] = a[i - 1] + a[i - 2];
    return a[n - 1];
}
```

When the for-loop is done, we return the last element of the array, which contains the requested F_n . Using this strategy, F_2 is calculated only once instead of the five times in the previous example. Note that because values are calculated only once, they are also *requested* less often: `a[1]`, which corresponds to F_2 , is used only twice instead of five times. However, because the calculation is performed bottom-up, writing the solution this way is often less intuitive than the D&C top-down way, which makes it harder to design and maintain. Also, we have to know in advance exactly which values we are going to need in our calculation of the final answer. For the Fibonacci sequence, this is not very hard: all Fibonacci numbers from F_1 to F_n are needed. For other problems, this can be less trivial, as we will see later on. If we cannot predict what values we will need, our DP solution will actually calculate too much, which might have a significant impact on the running time.

We now have a much more efficient algorithm compared to the one in the previous section. This implementation uses $O(n)$ space (the array) and $O(n)$ time (the for-loop to fill the array).

2.3 Memoization

In section 15.3, on page 347, [CLRS01] describes an variation of dynamic programming called *memoization*. Memoization combines the intuitive top-down calculation of D&C with the optimised running time of our DP approach. It does this by *caching* solutions to problems once they have been calculated. Then, when they are needed again, they are returned from the cache instead of being recomputed.

In our case, we are going to use an array to cache the values, just like in the previous approach. Let's take a look at the code.

```
// The cache. Assume it has been
// created and has sufficient size.
int[] a;

int fib(int n) {
    if (a[n - 1] == 0)
        a[n - 1] = doFib(n);           <-- 2
    return a[n - 1];                  <-- 1
}

int doFib(int n) {
    if (n == 1 || n == 2)
        return 1;
    else
        return fib(n - 1) + fib(n - 2); <-- 3
}
```

There are three labels in the code:

1. When `fib` is asked for F_n , in most cases it will be able to return the value from the cache right away.
2. The first time `fib` is asked for a specific Fibonacci number, however, we have to compute the value. We do this by calling `doFib`. Because in Java arrays are initialised with zeroes and all our Fibonacci numbers are positive, we know F_n has not been computed yet if and only if `a[n - 1]` is still zero.
3. `doFib(n)` actually computes F_n . Note that when `doFib` needs smaller Fibonacci numbers, it calls `fib`, not `doFib`.

Figure 2 shows a tree representing the method calls for this algorithm. As you can see, a significant part of the tree from figure 1 is cut away and only the left side remains. Despite the slight overhead due to the memoization, this algorithm is $O(n)$ in both space and time, just like the DP approach. And like the D&C approach, it is defined in an intuitive, top-down manner.

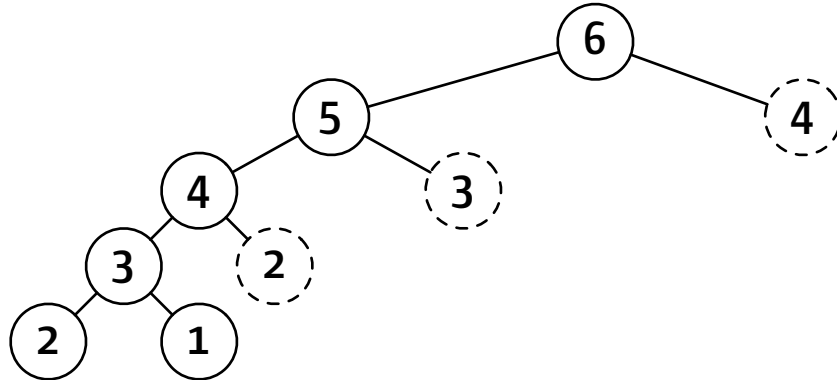


Figure 2: The method call tree for the memoization approach to calculating Fibonacci numbers. Nodes with dashed strokes are the memoized values.

It is interesting to note that, because functions in functional programming languages are free of side effects, one could implement standard memoization in such languages. [Hug85] In that case, the mathematical definition of the Fibonacci sequence could be transformed directly into, for example, Haskell, still yielding a $O(n)$ time-complexity algorithm:

```

fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)
  
```

However, according to Jeroen Fokker, this isn't implemented in modern Haskell compilers because it imposes a significant cost on the running time (a check every time a function is called) and space usage (maintaining the cache). It is only effective in cases where a function is actually called repeatedly with the same values as arguments, and finding out which these functions are at compile-time (out of the many functions a complete module usually contains) is probably very complicated. Perhaps the programmer could help the compiler by attaching flags (using a keyword, for example) to to-be-memoized functions. Then again, one could argue that this stimulates laziness in programmers.

Memoization is very valuable if fully filling the data structure from bottom to top in a DP manner would actually be doing too much work, such as in the binary knapsack problem (not discussed in this article). It is also useful if you are not sure beforehand in which order the subproblems need

to be computed, because the solutions to the subproblems are computed ‘lazily’: when needed, and not any earlier.

2.4 A constant-space algorithm

Although it doesn’t have much to do with dynamic programming, I feel this section wouldn’t be complete if I kept one final Fibonacci implementation from the reader. This approach is often used in books on programming languages, such as *The Java Programming Language, Second Edition* [ABGJ98], to demonstrate the use of variables and loops.

So far we have always remembered all previously computer Fibonacci numbers. This isn’t necessary; to compute F_n you only need to know the previous two numbers. That is exactly what this implementation does. It uses two variables: `lo` containing F_i and `hi` containing F_{i+1} . At the next iteration, the indices are incremented by one, causing `lo` to contain F_{i+1} and `hi` to contain F_{i+2} . This is possible because the new value of `hi` can be expressed as a function of the two previous Fibonacci numbers. You might think that you need a third, temporary variable to contain the new F_{i+2} while the values of `lo` and `hi` are updated, but with a small trick this isn’t necessary. We continue iterating until `hi` contains the requested F_n .

```
int fib(int n) {
    int lo = 1; // fib 1
    int hi = 1; // fib 2

    for (int i = 2; i < n; i++) {
        // At the moment, lo = fib (i - 1) and hi = fib i.

        hi = lo + hi; // Set hi to fib (i - 1) + fib i = fib (i + 1)
        lo = hi - lo; // Set lo to fib (i + 1) - fib (i - 1) = fib i
    }

    // At the last iteration, i = n - 1, which means hi
    // now contains the requested fib (n - 1 + 1) = fib n.

    return hi;
}
```

Note that for $n = 1$ `fib` actually returns F_2 . This is no problem, however, because $F_1 = F_2$. This algorithm uses $O(1)$ space and $O(n)$ time.

Yet another implementation is described by HAKMEM, item 12 [BGS72]. It claims to have $O(\log n)$ time complexity and uses something called *pair-wise multiplication*.

3 Ingredients of dynamic programming

It seems there are as many definitions of ‘dynamic programming’ as there are computer scientists. [CLRS01] makes a strict distinction between dynamic programming and memoization. Searching for dynamic programming with Google however, it seems that memoization is often seen as a form of dynamic programming. The Wikipedia article on Dynamic Programming [Wik05], for example, says:

Dynamic programming usually takes one of two approaches:

- **Top-down approach:** The problem is broken into subproblems, and these subproblems are solved and the solutions remembered, in case they need to be solved again.
- **Bottom-up approach:** All subproblems that might be needed are solved in advance and then used to build up solutions to larger problems.

Obviously, the ‘top-down approach’ of dynamic programming is the approach described in section 2.3, while the ‘bottom-up approach’ of dynamic programming is the one described in section 2.2. In the rest of this article, I continue to make the distinction as [CLRS01] does.

Then there is still the question of what the definition of dynamic programming is. [CLRS01] does not give a definition; rather, they give two properties of problems solvable with dynamic programming that most sources, including Wikipedia, seem to agree with, even though Wikipedia fails to refer to [CLRS01]. These two properties are:

- **Optimal substructure.** As said before, the solution to the problem should be expressible as a function of solutions to smaller problems: a recursive definition. This is often the most difficult and important step, as we will see in later examples. The inductive hypothesis is crucial here: assume the small problems’ solutions are known and build the solution to the current problem from that. Then the only thing left to do is define the answers to the elementary problems separately.
- **Overlapping subproblems.** Recall the first Fibonacci algorithm we discussed: we were repeatedly calculating the same Fibonacci numbers. This was the very reason to apply dynamic programming; if there are no overlapping subproblems, a simple recursive implementation will suffice.

Once you have recognised that the problem has these two properties, there are some other things you have to think about before you can actually construct an algorithm:

- **Data structure.** What data structure are you going to use? For the Fibonacci sequence, we used a 1-dimensional array. A 2-dimensional table however is much more common, as we will see in later examples. There might be problems for which you need a 3-dimensional table, a map or another structure entirely.
- **Indexing.** Closely related to the data structure is the indexing: you need to find indices or names for the different subsolutions so you can address them and use them in expressions. Suppose your data structure is a table. Another way to look at the indexing question is to ask yourself: what is the meaning of the value in cell (i, j) of my table? Writing down a clear answer to this question helps much in understanding the algorithm. It is also important to think about the order of the columns and rows in the table. Do they matter? For the Levenshtein distance problem (discussed in the section below), they are. For the binary knapsack problem, the order is relevant in one direction and irrelevant in the other – for the correctness of the solution, that is.

4 Levenshtein distance

A very useful application of dynamic programming is the problem of the *Levenshtein distance* or *edit distance* of two strings. It is closely related to the Hamming distance. In fact, it can be seen as a generalisation of the Hamming distance.

The Hamming distance is defined for two strings s and t of equal length n , and is the number of positions i , $0 \leq i < n$, for which $s_i \neq t_i$ (where s_i is the character at position i in s). For example, the Hamming distance between “butter” and “flower” is 4. Another way to look at this is the minimum number of character substitutions needed to transform s into t .

When calculating the Levenshtein distance, two other operators besides substitution are available: addition and deletion of a character. This allows the Levenshtein distance to be defined for two strings of unequal length too. For example, the Levenshtein distance between “coefficient” and “efficiency” is 4: delete the ‘c’ and ‘o’, change the ‘t’ into a ‘c’ and add a ‘y’.

The Levenshtein distance and variants have many practical uses, including spelling checkers, fraud detection for students, speech recognition and DNA sequence aligning.

Let $d(s, t)$ be the Levenshtein distance between t and s . How can we implement an algorithm that computes this value? The first thing to notice is that if t is the empty string ε , all we need to do is delete all characters from s , one by one. This takes $|s|$ operations, so $d(s, \varepsilon) = |s|$. Because d is symmetrical (i.e. $d(s, t) = d(t, s)$ for all strings s and t), a similar reasoning

goes if s is empty: $d(\varepsilon, t) = |t|$. If both s and t are non-empty, then we can consider the three different operations. Let's take $s = \text{"coefficient"}$ and $t = \text{"efficiency"}$ as our examples again.

- **Deletion.** We might be able to transform "coefficient" into "efficiency" by first removing the "t" and then transforming "coefficient" into "efficiency". The number of operations required to do this is $d(\text{"coefficient"}, \text{"efficiency"}) + 1$.
- **Insertion.** Another option is to first transform "coefficient" into "efficiency" and then inserting (appending) the "y". The number of operations needed is $d(\text{"coefficient"}, \text{"efficiency"}) + 1$.
- **Substitution.** The last operation possible is to substitute the "t" by a "y". Then we still need to transform the rest of the strings, and the total number of operations is $d(\text{"coefficient"}, \text{"efficiency"}) + 1$.

We simply choose that operation which minimizes the required number of operations. This means that we have found a recursive definition of d :

$$d(\varepsilon, t) = |t| \tag{1}$$

$$d(s, \varepsilon) = |s| \tag{2}$$

$$d(s, t) = \min \begin{cases} d(\text{sub}(s, 1, |s| - 1), t) + 1 & \text{(deletion)} \\ d(s, \text{sub}(t, 1, |t| - 1)) + 1 & \text{(insertion)} \\ d(\text{sub}(s, 1, |s| - 1), \text{sub}(t, 1, |t| - 1)) + 1 & \text{(substitution)} \end{cases} \tag{3}$$

$\text{sub}(s, a, b)$ is a function that returns a substring of s from position a to b inclusive. Note that $\text{sub}(s, a, a - 1)$ yields the empty string ε . A simple D&C implementation of the recursive definition would do far too much work because of overlapping subproblems, similar to the D&C approach to the Fibonacci sequence. Therefore, we are going to cache the results again, in a $(|s| + 1, |t| + 1)$ table where cell (i, j) , i and j 0-relative, is set to $d(\text{sub}(s, 1, i), \text{sub}(t, 1, j))$. After filling the table bottom-up, cell $(|s|, |t|)$ contains the needed $d(s, t)$.

Now we have an algorithm that is $O(|s| \times |t|)$ in time and space that yields $d(s, t)$. In most cases (in the case of spelling checkers, for example), this is enough. Sometimes, however, we also want to know what the actual operations are to transform t into s . This is fairly easy to figure out; cell (i, j) contains the minimum of three of the surrounding cells, each of those three cells corresponding to one of the three operations. Depending on which of the surrounding cells equals cell (i, j) , we know what operation is preferred at that particular subproblem. If there is more than one equal surrounding cell, then we have a tie, which means there are several distinct sets of $d(s, t)$ operations all yielding the same final transformation.

5 Other examples

In this article we have seen two problems we solved using dynamic programming: the Fibonacci sequence to demonstrate the various approaches, and the Levenshtein distance which is a very common problem. In this section I will list a few other problems solvable with dynamic programming.

- The **matrix-chain multiplication** problem. Although different parenthesizations of the matrix product $A \times B \times C$ yield the same result, the number of multiplications and additions needed for the several options can differ significantly. Therefore, it pays off to first determine the optimal parenthesization before actually computing the result. This can be done efficiently using DP.
- The **binary knapsack** problem. This is a classical problem about finding an optimal set of items, each with their own value and weight, that fit in a knapsack with limited capacity.
- The **Cocke-Younger-Kasami** algorithm determines whether a given string can be generated by a given context-free grammar.
- The **Viterbi** algorithm is used in the context of *hidden Markov models* and has applications in natural language processing and speech recognition.
- The **longest common subsequence** of two strings. This problem is closely related to the Levenshtein distance.

6 Conclusion

Dynamic programming is a powerful way to significantly reduce the running time complexity of algorithms to specific problems. Memoization may prove a valuable alternative if the dependencies are not well-known beforehand. If there are no overlapping subproblems, a simple recursive implementation suffices.

References

- [ABGJ98] Ken Arnold, Gilad Bracha, James Gosling, and Bill Joy. *The Java programming language (2nd edition)*. Addison-Wesley Publishing Co., New York, NY, USA, 1998.
- [BGS72] Michael Beeler, R. W Gosper, and Rich Schroeppel. Hakmem. Technical report, Cambridge, MA, USA, 1972.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2001.
- [Hug85] John Hughes. Lazy memo-functions. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 129–146, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [Wei99] Eric W. Weisstein. Fibonacci number. *MathWorld*, 1999.
- [Wel86] D. Wells. *The Penguin Dictionary of Curious and Interesting Numbers*. Penguin Books, Middlesex, England, 1986.
- [Wik05] Wikipedia. Dynamic programming, 2005. Online; accessed 08-November-2005.