Olympiads in Informatics, 2015, Vol. 9, 45–55
DOI: http://dx.doi.org/10.15388/ioi.2015.05

Towards a Better Way to Teach Dynamic Programming

Michal FORIŠEK
Comenius University, Bratislava, Slovakia
e-mail: forisek@dcs.fmph.uniba.sk

Abstract. We give an overview of how various popular algorithm textbooks deal with the topic of dynamic programming, and we identify various issues with their expositions. In the second part of the paper we then give what we believe to be a better way of presenting the topic. While textbooks only contain the actual exposition, our paper also provides the rationale behind our choices. In particular, we managed to divide the topic into a sequence of simpler conceptual steps that are easier to learn.

Keywords: algorithms, dynamic programming, memoization, task analysis.

1. Overview

Dynamic programming is a standard paradigm used in the design of efficient algorithms. This approach is usable for problems that exhibit an optimal substructure: the optimal solution to a given instance can be recursively expressed in terms of optimal solutions for some sub-instances of the given instance.

Dynamic programming comes in two basic flavors. The top-down approach, usually called *memoization*, is based on implementing the computation of the discovered recursive relation as a recursive function, and then adding a cache so that each sub-instance only gets evaluated once. The bottom-up approach, usually called *dynamic programming*, essentially evaluates the same recurrence but in an iterative way: the algorithm designer specifies an order in which the sub-instances are processed, and this order is chosen in such a way that whenever we process a particular instance, all its needed sub-instances have already been processed and their optimal solutions are already known by the algorithm.

Below, we use the term *dynamic programming (DP)* to cover both flavors. When talking specifically about the iterative approach we will use the term *iterative DP* or *bottom-up DP*.

Despite being conceptually easy, dynamic programming is notorious for being hard to learn. Quoting Skiena (2008): "[Until] you understand dynamic programming, it seems like magic."

Different textbooks use very different approaches to present dynamic programming. The canonical way of presenting dynamic programming in algorithm textbooks is by showing a sequence of tasks and solving them using dynamic programming techniques. What is usually missing is:

- Rationale for choosing these specific tasks and their order.
- Notes on potential pitfalls when presenting the tasks and their solutions.

In this paper we aim to fill in those missing gaps. More precisely, the paper consists of the following parts:

- We present the way dynamic programming is exposed in multiple standard algorithm textbooks.
- We analyse those expositions and identify a set of possible pitfalls that often confuse and mislead students.
- We present our suggested version of a better order in which to teach the individual concepts related to dynamic programming, and we argue about the benefits of our approach.

2. Algorithm Textbooks

Throughout this paper we are going to refer to the way dynamic programming is treated in some of the canonical algorithm textbooks. In particular, we examined the following ones: Cormen *et al.* (2001), Dasgupta *et al.* (2006), Kleinberg and Tardos (2006), Sedgewick (1998), Skiena (2008). When refering to these textbooks below, for better readability we will use the following shorthand instead citations: CORMEN, DASGUPTA, KLEINBERG, SEDGEWICK, and SKIENA.

Below we give a brief summary how each of these textbooks introduces dynamic programming.

CORMEN prefers and almost exclusively uses a bottom-up approach. Dynamic programming is introduced using the following sequence of tasks and texts:

1. Assembly-line scheduling.
2. Matrix chain multiplication.
3. A general overview of iterative dynamic programming and memoization.
4. Longest common subsequence.
5. Optimal binary search tree.

DASGUPTA also prefers and almost exclusively uses a bottom-up approach, on a rather large set of solved tasks:

1. Shortest paths in a DAG.
2. Longest increasing subsequence.
3. A warning against exponential-time recursion.
4. Edit distance.
5. Knapsack.
6. A note about memoization.
7. Matrix chain multiplication.

8. Shortest paths.
9. Independent sets in trees.

KLEINBERG first introduces dynamic programming using a top-down approach, but then uses a bottom-up iterative approach in all following problems. The entire exposition looks as follows:

1. Weighted interval scheduling.
2. A general overview of iterative dynamic programming and memoization.
3. Segmented least squares.
4. Subset sums and Knapsack.
5. RNA secondary structure.
6. Sequence alignment (and optimizations to reduce memory use).
7. Shortest paths.

SEDGEWICK prefers a top-down approach. Only presents two problems:

1. Fibonacci numbers.
2. Knapsack.

Note that the new 4th edition (Sedgewick and Wayne, 2011) no longer contains a section on dynamic programming.

SKIENA is also in favor of starting with the top-down approach. His exposition proceeds in the following order:

1. Fibonacci numbers: recursively, with memoization, iteratively.
2. Binomial coefficients.
3. Edit distance.
4. Longest increasing subsequence.
5. Linear partition.
6. Context-free grammar parsing.

3. Critical Analysis

In this section we show the results of our analysis of the expositions used in the textbooks mentioned above. We mostly focus on tasks used in multiple textbooks.

3.1. *Matrix Chain Multiplication*

Statement: Given is a sequence $M_1, \ldots, M_n$ of rectangular matrices such that the product $M_1 \times \cdots \times M_n$ can be computed. Clearly, it can be computed as a sequence of $n-1$ standard matrix multiplications, and the result does not depend on their order. Given the assumption that multiplying an $a \times b$ and a $b \times c$ matrix takes $\Theta(abc)$ time, what is the most efficient way of computing the entire product?

The problem is solved by dynamic programming over all intervals. I.e., the states can be described by pairs of indices i, j such that $i \leq j$. For each state we compute the best

solution for the matrices in the given range.

Issues with this problem:

- Incomprehensible to students who lack background in linear algebra. The problem feels unnatural and the cost function seems arbitrary.
- Unnecessary clutter: the input is a sequence of ordered pairs of integers. There are other similar problems on integer sequences and/or strings.
- Lack of practical motivation. Finding a clear practical application for this algorithm is probably impossible.
- The existence of a much better solution. The $\Theta(n^3)$ DP algorithm shown in textbooks is an overkill, Hu and Shing (1982, 1984) gave a different $O(n \log n)$ solution for this problem.

3.2. *Shortest Paths in DAGs*

Statement: Given is a weighted directed acyclic graph (DAG). Find the shortest path from vertex 1 to vertex n.

This is a very good problem to be used at some point during the instruction on dynamic programming – mostly because it is the most general one. Essentially all dynamic programming solutions can be viewed as computations on directed acyclic graphs: the states of the computation (i.e., sub-instances we are solving) are the vertices of the DAG, the recursive relation determines the edges, and the order in which an iterative DP solution evaluates the states must correspond to a topological order of this graph.

Issues: DASGUPTA uses this problem as the first problem on which a dynamic programming approach is presented. We strongly advise against that. While we agree that the concepts mentioned in the previous paragraph are important, we believe that the proper time and way to learn them is by abstraction after already being familiar with many specific problems solved using dynamic programming.

Additionally, this problem requires students to be able to store and access a graph, and the data structures needed to do so efficiently are more involved than simple static arrays. A detailed analysis of the time and space complexity is also non-trivial as there are two different parameters (the number of vertices and the number of edges). This is especially true if one tries to solve this problem using recursion without memoization.

3.3. *Longest Common Subsequence*

Statement: Given two sequences (or strings), find one longest sequence that occurs as a (not necessarily contiguous) subsequence in each of them.

Related problems: Edit distance (Levenshtein distance) between two strings, DNA sequence alignment.

This is, and certainly should be, the gold standard among introductory problems solvable using dynamic programming. The solution only requires basic arrays, the sub-

problems and the recurrence are natural, and each of the subproblems can be evaluated in constant time.

Later, this problem can be used when discussing the differences between the top-down and the bottom-up approach.

This problem also leads to an advanced topic: Hirschberg's implementation (Hirschberg, 1975) of a bottom-up DP solution that can reconstruct an optimal solution in linear memory.

Issues: The time complexity of the brute force approach (recursive search without memoization) is hard to analyse exactly and it is often neglected in textbooks. CORMEN only mentions it to be "exponential-time" without any details, while DASGUPTA and KLEINBERG completely avoids mentioning it. SKIENA is the only one to address it, showing a (non-tight) 3^n lower bound for his version of the Edit distance problem.

3.4. *0–1 Knapsack*

Statement: Given is an integer weight limit W and a collection of n items, each with an integer weight w_i and an arbitrary cost c_i. Find a subset of items that has a total weight not exceeding the given limit and the largest possible total cost.

Related problems: Knapsack where arbitrarily many copies of each item are available. Coin change problems.

This is a reasonably natural class of problems. Again, their advantage is that the implementation only requires basic tools and that the recurrence relation is simple.

Issues: The whole notion of pseudopolynomial time. At some point, the students need to be explained why an algorithm that runs in $O(nW)$ is not considered a polynomial-time algorithm. While this issue is orthogonal to the concept of dynamic programming, it is an inherent part of this task and it should come up during its analysis. From experience, this may be the most challenging part of the problem for the students.

SEDGEWICK avoids the topic of pseudopolynomial time completely. It is just mentioned that the algorithm is only useful if the capacities are not huge. KLEINBERG also avoids this topic completely.

DASGUPTA addresses the topic with a single brief note: "[...] they can both be solved in $O(nW)$ time, which is reasonable when W is small, but is not polynomial since the input size is proportional to $\log W$ rather than W."

3.5. *Fibonacci Numbers*

Statement: Given n, compute the n-th Fibonacci number.

Fibonacci numbers are an excellent source of what is possibly the simplest non-trivial recurrence relation. They can easily be used to demonstrate the effect of memoization, as a straightforward recursive function that computes their values runs in exponential time.

Issues: The only issue with this very simple problem is that the numbers themselves grow exponentially and their values quickly exceed the range of standard integer variables in most languages. And even if you use a programming language with arbitrary precision integers (e.g., Python), the size of these numbers plays a role in estimating the time complexity of efficient programs.

A common way to address this issue is to modify the problem: instead of computing the exact value of the n-th Fibonacci number we aim to compute its value modulo some small integer. (E.g., if the modulus is 10^9, we are in fact computing the last 9 decimal digits of F_n.) Here we would just like to remark that the Fibonacci sequence modulo any m is necessarily periodic and this observation leads to asymptotically more efficient algorithms.

4. Our Approach to Teaching Dynamic Programming

In this final section we give a detailed presentation of how we suggest to teach dynamic programming. For each task used we clearly state and highlight the new concepts it introduces, and we argue why our way of introducing them works.

Note that we intentionally start with the top-down version of dynamic programming, i.e., by adding memoization to recursive functions. This is intentional and very significant. The main purpose of this choice is to show the students how to break up the design of an efficient solution into multiple steps:

1. Implement a recursive algorithm that examines all possible solutions.
2. Use the algorithm to discover a recursive relation between various subproblems of the given problem.
3. Add memoization to improve the time complexity, often substantially.
4. Optionally, convert the solution into an iterative bottom-up solution.

The more traditional approach that starts with iterative DP requires students to do steps 2 and 4 at the same time, without giving them good tools to do the analysis and to discover the optimal substructure. In our approach, step 1 gives them such a tool: once we have the recursive solution, the arguments of the recursive function define the subproblems, and we can examine whether the function gets called multiple times with the same arguments. If it does, we know that the problem does exhibit the optimal substructure, and in step 3 we mechanically convert our inefficient solution into an efficient one.

Lesson 1: Fibonacci numbers

Goals: Observe a recursive function with an exponential time complexity. Discover the source of inefficiency: the function executes the same recursive call many times.

Fibonacci numbers have a well-known recurrence: $F_0 = 0$, $F_1 = 1$, and $\forall n > 1 : F_n = F_{n-1} + F_{n-2}$. In our presentation we use the following Python implementation:

```
def F(n):
  if n==1 or n==2:
    return 1
  else:
    return F(n-1) + F(n-2)
```

Note that this implementation neglects the case $n = 0$ and uses $n = 1$ and $n = 2$ as the base case. This is intentional, the purpose is a more elegant analysis later.

We can now run this program and have it compute consecutive values of the Fibonacci sequence:

```
for n in range(1,100): print( n, F(n) )
```

The first few rows of input will appear instantly but already around $n = 35$ the program will slow down to a crawl. We can empirically measure that each next value takes about 1.6 times longer to compute than the previous one.

What is going on here? The easiest way to see it is to log each recursive call:

```
def F(n):
  print('calling F(' + str(n) + ')')
  ...
```

Already for small values like $n = 6$ we quickly discover that the same function call is made multiple times. Here it is instructional to show the entire recursion tree for $n = 6$, we omit the picture here to conserve space.

Here, a suitable homework is to leave the students analyse how many times $F(n - k)$ gets called during the computation of $F(n)$. For our version of the implementation the answer to this question are again precisely the Fibonacci numbers.

Alternately, we can just directly estimate the whole time complexity: when computing $F(n)$, each leaf of the recursion tree contributes 1 to the final result.

(This is the rationale for our choice to use $n = 1$ and $n = 2$ as base cases.) Hence, there are precisely $F(n)$ leaves and thus precisely $F(n) - 1$ inner nodes in the recursion tree. In other words, the *running time* of the computation of $F(n)$ is clearly proportional to the *value* of $F(n)$, which is known to grow exponentially.

Lesson 2: Memoization

Goals: Learn about memoization and conditions when it can be applied.

One of the points that is woefully neglected in traditional textbooks is the difference between *functions* in the mathematical sense and in the programming sense. The output of a mathematical function only depends on its inputs: $\cos(\pi/3)$ today is the same value as $\cos(\pi/3)$ tomorrow. For a function in a computer program, two consecutive calls with the same arguments may often return different values. There are lots of different reasons why this may happen. For instance, the output of the function may depend on global variables, on environment variables (such as the current locale settings), on pseudorandom numbers, on the input from a user, etc. (Listing these is actually a lovely exercise for students!)

Given the above observation, memoization is a very straightforward concept for students: we simply want to avoid computing the same thing twice. And it should now be clear that any function in our program that is also a function in the mathematical sense can be memoized. (Such functions are sometimes called *pure functions*.)

An interesting historical note: memoization is not only useful when it comes to improving the asymptotic time complexity. For instance, many early programs that produced computer graphics used precomputed tables of sines and cosines because table lookup was faster than the actual evaluation of a floating-point-valued function.

Lesson 3: Fibonacci numbers revisited

Goals: See the stunning effect memoization can have.

By applying memoization (using a simple array) to the Fibonacci function, each of the values $F(1)$ through $F(n)$ is only computed once, using a single addition. Therefore, we suddenly have a program that only performs $\Theta(n)$ additions to compute the value $F(n)$: quite an improvement over the original exponential time. The above Python program can now easily compute $F(1000)$.

(Note that Python operates with arbitrarily large integers. The n-th Fibonacci number is exponential in n and therefore has $\Theta(n)$ digits. In the RAM model, the actual time complexity of the above algorithm is $O(n^2)$, as each addition of two $O(n)$-digit numbers takes $O(n)$ steps.)

Here it is important to highlight the contrast: exponential time *without* vs. polynomial time *with* memoization. It is also instructional to draw a new, collapsed version of the entire recursion tree for $n = 6$.

Lesson 4: Maximum weighted independent set on a line

Goals: Encounter the first problem solvable using dynamic programming. Learn how to write a brute force solution in a good way, and how to use memoization to "magically" turn it into an efficient algorithm.

Statement: Given is a sequence of n bottles, their volumes are v_0 through v_{n-1}. Drink as much as you can, given that you cannot drink from any two adjacent bottles.

This problem has a very short and simple statement and only requires a simple one-dimensional array to store the input. But the main reason why we elected to use this as the first example will become apparent once we implement and examine a brute force solution for this problem.

Our goal is to implement a recursive solution that generates all *valid* sets of bottles and chooses the best among them. Such a recursive solution can be based on a simple observation: either we choose the last bottle or we don't. If we don't, we want to find the best solution from among the first $n - 1$ bottles.

If we do, we are not allowed to take the penultimate bottle and therefore we are looking for the best solution among the first $n - 2$ bottles.

```
v = [ 3, 1, 4, 1, 5, 9, 2, 6 ]
def solve(k):
```

```
''' returns the best solution for the first k bottles '''
if k == 1: return v[0]
if k == 2: return max( v[0], v[1] )
return max( solve(k-1), v[k-1] + solve(k-2) )
```

It should now be obvious that the time complexity of this program is exponential – in fact, the number of recursive calls needed to evaluate $solve(n)$ is precisely the same as the number of calls needed to evaluate $F(n)$ in our first lesson.

A key observation to make here is that `solve` can be considered a pure function. Even though it does access the global variable `v`, its contents remain the same throughout the execution of the program. Hence, we may apply memoization to `solve` in order to reduce the time complexity from $\Theta(\phi^n)$ to $\Theta(n)$.

(Note that `solve` can easily be turned into a true pure function if we pass a constant reference to `v` to `solve` as a second argument.)

Lesson 5: An iterative solution to the previous problem

Goals: Learn about the duality between the top-down and the bottom-up approach.

The subproblems solved by the memoized recursive solution can be naturally ordered by size. The same recurrence can now be used to write an iterative solution. A side-by-side comparison of both programs helps highlight parts that remained the same / only changed syntactically.

Optional lesson 6: Paths in a grid

Goals: Developing a bottom-up solution directly.

Statement: Given is a grid. Count all shortest paths along the grid from $(0,0)$ to (a,b).

The answer is obviously the binomial coefficient $\binom{a+b}{a}$, but the path-based point of view allows a natural formulation of a recurrence relation: Let $P(x,y)$ be the number of ways to reach (x,y). Each path that reaches (a,b) goes either through $(a-1,b)$ or through $(a,b-1)$. Hence, $P(a,b) = P(a-1,b) + P(a,b-1)$.

Statement: Now some grid points are blocked by some obstacles. Count all paths that go from $(0,0)$ to (a,b) in $a+b$ steps and avoid all obstacles.

The recurrence remains the same, only now the blocked grid points have $P(x,y) = 0$. It is instructional to solve small instances on this problem on paper – in fact, many of our students are familiar with this problem from earlier Math classes.

Lesson 7: Longest common subsequence

Goals: Investigating the differences between the top-down and the bottom-up approach.

For this problem, we first show the entire process. First, we show a recursive solution that generates all common subsequences, starting by comparing the last elements of both sequences. Then, we show that adding memoization improves this solution from

a worst-case exponential one into a solution that runs in $O(n^2)$.

Finally, we convert the solution into an equivalent iterative one.

Afterwards, we focus on the following points:

- Memory complexity. The iterative solution can easily be optimized to use $O(n)$ memory only, the recursive one cannot.
- Execution time. So far, iterative solutions were better as they didn't perform the additional work related to function calls. However, in this problem we can easily find inputs (e.g., two identical sequences) where the recursive solution outperforms the iterative one. The lesson here is that the top-down approach only evaluates the subproblems it actually needs, while the iterative approach doesn't know which subproblems will be needed later and thus it must always evaluate all of them.

Lesson 8: Longest increasing subsequence in quadratic time

Goals: Examining the first example where a subproblem isn't evaluated in constant time. Understanding how this is reected in the time complexity estimates. Most importantly, seeing that the subproblems don't have to be instances of the original problem.

Statement: Given a sequence of numbers, compute the length of its longest increasing subsequence.

In our opinion, this problem is one of the most important ones in teaching dynamic programming properly. When compared to previous problems, this one is much harder for beginners. Here's the main reason: *The subproblems we need to solve aren't actually instances of the original problem.*

This problem is used by DASGUPTA and SKIENA. However, DASGUPTA just reduces it to paths in a DAG without explicitly mentioning the conceptual step where we change the problem. SKIENA treats the problem properly: notably, asking the question "what information about the first $n-1$ elements of [the sequence] would help you find the solution for the entire sequence?" (Still, note that this question is factually incorrect: you are, in fact, supposed to look for information about the first $n-1$ elements that would help you find *the same information* for all n elements.)

We suggest actually emphasizing the redefinition of the problem. First, we illustrate on an example that knowing the length of longest increasing subsequence in the first $n-1$ elements is useless for solving the same problem for the first n elements. Only then we ask the question how to modify the problem in a way that would be useful. And the question is easily answered by using our approach: we can easily write a recursive solution that generates all increasing subsequences by choosing where to end and then going backwards. Converting this program into an $O(n^2)$ one requires just the mechanical step of adding memoization.

Optional follow-up lessons

After the above sequence of problems and expositions the students should have a decent grasp of the basic techniques and they should be ready to start applying them to new

problems. Still, there are more faces to the area of dynamic programming. Here, we suggest some possibilities for additional content of lectures:

- Problems where the state is a substring of the input: Edit distance to a palindrome. Optimal BST.
- Problems solvable in pseudopolynomial time: Subset sum, knapsack, coin change.
- Optimalizations of the way how the recurrence is evaluated: Hirschberg's trick for LCS in linear memory; Knuth optimization for Optimal BST; improving Longest increasing subsequence to $O(n \log n)$ by using a balanced tree to store solutions to subproblems.

5. Conclusion

Above, we have presented one possible way in which dynamic programming can be introduced to students. Our opinion is that the main improvement we bring is the systematic decomposition of the learning process into smaller, clearly defined conceptual steps.

References

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2001). *Introduction to Algorithms*. MIT Press, 2nd edition.

Dasgupta, S., Papadimitriou, C.H., Vazirani, U.V. (2006). *Algorithms*. McGraw-Hill.

Hirschberg, D. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6), 341–343.

Hu, T.C., Shing, M.T.(1982). Computation of Matrix Chain Products (part 1). *SIAM Journal on Computing*, 11(2), 362–373.

Hu, T.C., Shing, M.T. (1984). Computation of Matrix Chain Products (part 2). *SIAM Journal on Computing*, 13(2), 228–251.

Kleinberg, J., Tardos, É. (2006). *Algorithm Design*. Addison-Wesley.

Sedgewick, R. (1998). *Algorithms in C++*. Addison-Wesley, 3rd edition.

Sedgewick, R., Wayne, K. (2011). *Algorithms*. Addison-Wesley Professional, 4rd edition.

Skiena, S.S. (2008). *The Algorithm Design Manual*. Springer-Verlag, 2nd edition.

M. Forišek is an assistant professor at the Comenius University in Slovakia. Since 1999 he has been involved in organizing international programming competitions, including the IOI, CEOI, and ACM ICPC. He is also the head organizer of the Internet Problem Solving Contest (IPSC). His research interests include theoretical computer science (hard problems, computability, complexity) and computer science education.