Rathindra Nath Dutta

Design & Analysis of Algorithms

Contents

CHAPTER 1

Dynamic Programming

> *Those who cannot remember the past are condemned to repeat it.*

Dynamic Programming , *Internet*[1]

Dynamic Programming , like the Divide-and-Conquer or Greedy method is just another algorithm design tool. The term 'Dynamic Programming ' was coined by Richard Bellman and the naming has little to do with what it does rather has a interesting bit of history; interested readers may refer to appendix A. One generally uses Dynamic Programming for solving some *optimization problem*[2] whose brute-force solution takes exponential time with respect to the input size and we simply cannot use Divide-and-Conquer , Greedy or any other easy approach.

One might wonder whether Dynamic Programming is kind last resort i.e. go to tool when other techniques like Divide-and-Conquer or Greedy fails. Rephrasing this, one can see that essentially the question is: can Dynamic Programming solve any problem? The answer is obviously no. There are yet unsolvable problems like the classical Halting problem. Furthermore there are also solvable problems where Dynamic Programming cannot be applied! We will study a few such cases . So now another question arises: given a problem, how can one know whether Dynamic Programming is a suitable tool or not!?

To apply Dynamic Programming one must first verify whether the problem exhibits the two essential properties namely: *optimal substructure* and *overlapping subproblem*. Note that for a given optimization problem, designing an efficient (hopefully polynomial time) algorithm is all about carefully looking into the solution space and cautiously examining very few(should be polynomial on input size) of the feasible solutions and reporting the optimal one. A problem must possess aforementioned two properties in order to get real benefit from applying Dynamic Programming .

[1] `https://www.hackerearth.com/practice/notes/dynamic-programming-i-1`
`https://www.quora.com/What-does-dynamic-programming-tell-you-about-life`

[2] A optimization problem asks one to optimize(minimize or maximize) some quantity expressed by a objective function with respect to some given set of constraints.

Optimal Substructure

The optimal solution of the problem must be composed of optimal solution(s) of some subproblem(s). Whenever we combine optimal solutions of more than one subproblem, they must be independent of each other. In essence, we can define the optimal solution recursively in terms of optimal solution(s) of the subproblem(s).

Overlapping Subproblem

A recursive algorithm tries to solve same sub problem over and over again. Total number of distinct subproblems solved is relatively small, typically polynomial on input size.

You might be wondering how can Dynamic Programming rely on on subproblems being both independent and overlapping - do they not contradict each other? The answer is no, they actually express two different notion. Two subproblems of the same problem are independent if they do not share resources. Therefore, they could easily be combined together to form a candidate solution. On the other hand, we say two subproblems are overlapping if they are really the same but occurs as subproblems of two different problems.

Let us first understand how these two actually helps in designing an efficient Dynamic Programming algorithm. Typically there are multiple independent subproblems which may correspond to various candidate solution of the original one. But not all of them leads to the optimal solution. If we somehow knew(say lucky guess) which subproblem(s) to pick, all it remains to build upon the optimal solution(s) of it(them). But there is no such thing as lucky guess in algorithm design! The first property actually helps us narrow down the candidate solution sets of a subproblem, as we only need to consider the optimal one. We can now practically apply brute force on this reduced search space and pick the subproblem(s) that actually leads to the optimal solution. Whereas the second property implies that is a recursive algorithm revisits same subproblem repeatedly, we can cache the previously computed result into some table. Thus we are actually computing only once, the first time; all subsequent requirements can be served by a constant time table lookup. This technique is called *memoization*. It might seem a little abstract, because it is, but this will become clear as we explore a few Dynamic Programming algorithms in the later sections.

Essentially a dynamic programming solutions for different problems varies on two things: how many distinct subproblems are actually needed to consider to get the optimal solution, and how many choices are there for determining which subproblem(s) to use in an optimal solution. Informally, the running time of a Dynamic Programming algorithm depends of these two factors.

Dynamic Programming algorithms can be formulated in two ways; either by a **top-down** recursive approach with memoization or using a **bottom-up** approach where we optimally solve smaller subproblems first then gradually increase the problem size until

we get optimal solution of the original problem. In general, if all subproblems must be solved at least once, a bottom-up strategy usually outperforms the corresponding top-down memoized algorithm by a constant factor. The top-down approach has overheads for the recursive calls and table maintenance. Alternatively, if some subproblems in the subproblem space need not to be solved at all, the memoized top-down version has the advantage to solve only the required ones.

Designing a Dynamic Programming algorithm more or less involves these four steps:

1. Characterize the structure of the optimal solution
2. Recursively define the value of the optimal solution
3. Compute the value of the optimal solution; typically in bottom-up fashion
4. Reconstruct (if required) the optimal solution from the choices made during step 3

In this chapter we will start by comparing Dynamic Programming with Divide-and-Conquer and Greedy followed by the implementation details of *memoization* and finish with a few case studies.

1.1 Dynamic Programming vs Divide-and-Conquer vs Greedy

Both Dynamic Programming and Divide-and-Conquer approach breaks the problem into subproblems and build the solution based on the solution of the subproblems. On the other hand both Dynamic Programming and Greedy rely on having optimal substructure property of the problem definition. Therefore it is intuitive compare these three strategies and know their differences.

In general, a problem having an efficient Divide-and-Conquer algorithm often lacks the overlapping subproblem property. In Divide-and-Conquer algorithms at each recursive call it generates new and new subproblems to solve; therefore memoization does not help whereas in Dynamic Programming same subproblems are revisited repeatedly and memoization does help significantly.

Although both the Greedy strategy Dynamic Programming exploits optimal substructure property one major difference is how they choose a subproblem. Instead of evaluating all candidate subproblems and choosing the one that leads to a globally optimal solution, a Greedy algorithm first makes a ‘greedy’ choice - that seems best at the moment, only then it tries to solve the chosen subproblem. Therefore Greedy algorithms are often faster but there is the risk of being stuck at local optima. One can think of Dynamic Programming as bottom-up approach where we choose suitable subproblems at each step and finally get the optimal solution, whereas Greedy works in top-down fashion

where at each step it makes a greedy locally optimum choice and then only solves the resulting subproblem.

1.2 *Memoization*

Before delving further into Dynamic Programming paradigm let us first study how to implement memoization and how much benefit we can actually obtain. Consider the problem of calculating the n-th Fibonacci number recursively as per the definition[3].

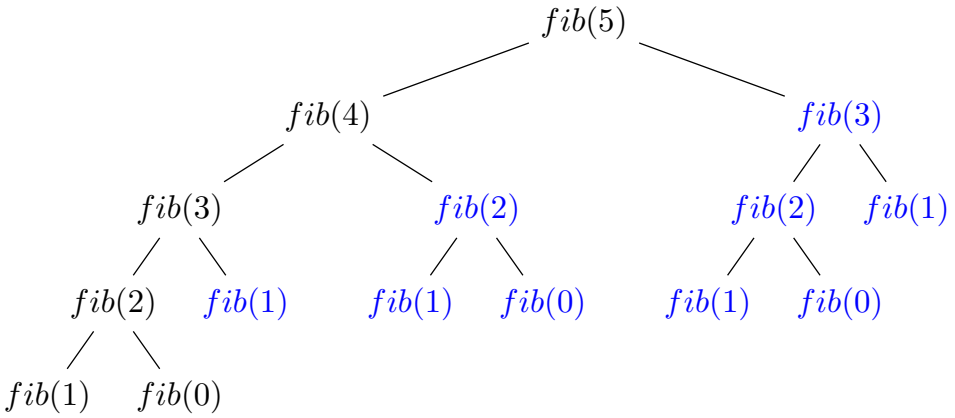

Figure 1.1: Recursion tree for $fib(n)$ when $n = 5$

Theorem 1.2.1. *There are $\Theta(2^n)$ recursive calls for a single call to $fib(n)$.*

Proof. Let $T(n)$ be the total number of recursive calls made for a single call to $fib(n)$. We will show that $2^{n/2} \leq T(n) \leq 2^n, \forall n \geq 2$ using induction on n.
Basis: We have $T(2) = 1 + 1 = 2$ therefore $2^{2/1} \leq T(2) \leq 2^2$. Similarly, $T(3) = T(2) + 1 = 3$ thus $2^{3/1} \leq T(3) \leq 2^3$.
Hypothesis: Let's assume that the statement holds for both $T(n-1)$ and $T(n-2)$.
Inductive step:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\leq 2^{n-1} + 2^{n-2} \\ &= 2^{n-2}[2+1] \\ &< 2^{n-2} \times 4 = 2^n \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\geq 2^{(n-1)/2} + 2^{(n-2)/2} \\ &= 2^{(n-2)/2}[\sqrt{2}+1] \\ &> 2^{(n-2)/2} \times 2 = 2^{n/2} \end{aligned}$$

■

[3] $fib(n) = \begin{cases} n & when\ n \leq 1 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$

As depicted in figure 1.1 there are exactly n recursive calls we need to compute the respective value for the first time; all subsequent calls, shown in blue, can be replied with the cached results. Caching return values in case of recursive calls is called memoization. The result is cached into a memo table indexed by the call argument(s). The following pseudo code should clarify how one can apply memoization. As evident from figure 1.1

Algorithm 1 Memoized Fibonacci Computation

```
1:  procedure INITTABLE(n)
2:      for i = 0 to n − 1 do
3:          f[i] ← −1
4:      end for
5:  end procedure
1:  procedure FIB(n)
2:      if f[n] ≠ −1 then return f[n]          ▷ return the memoized value
3:      end if
4:      if n ≤ 1 then                          ▷ base case
5:          f[n] ← n
6:          return f[n]
7:      end if
8:      f[n] ← FIB(n − 1) + FIB(n − 2)
9:      return f[n]
10: end procedure
```

this pseudocode runs in linear time as its only computes the value for the left branch from the root(the black nodes) and all right branch(the subtrees at rooted at some blue node) values are looked up in constant time from the table f. The recursion tree essentially gets reduced to figure 1.2.

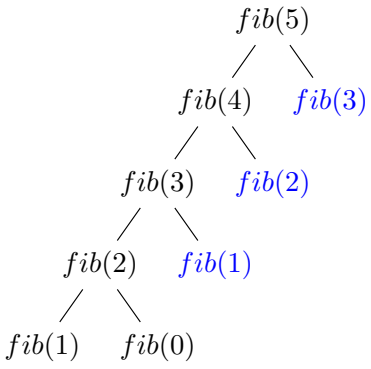

Figure 1.2: Memoized Recursion tree for $fib(n)$ when $n = 5$

Now consider the problem of calculating binomial coefficient recursively as follows:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & when\ 0 < k < n \\ 1 & otherwise \end{cases} \tag{1.1}$$

Theorem 1.2.2. *Show that the above expression correctly computes* $\binom{n}{k}$

Proof. Left as an exercise. ■

Theorem 1.2.3. *Show that a simple recursive formulation takes* $\Omega(\binom{n}{k})$ *time.*

Proof. Left as an exercise. ■

We can use memoization here also; the memo table in this case generates the Pascal's triangle as depicted in figure 1.3. The table is filled line by line. In fact, it is not even required to store a matrix; it is sufficient to just keep a vector of length k for the current line and calculate the next line by updating the entries from left to right. Thus the algorithm takes $O(nk)$ time and $O(k)$ space for its execution. A practice exercise would be to ask you to formally write the algorithm (either in plain text or in pseudocode format).

	0	1	2	...	k-1	k
0	1					
1	1	1				
2	1	2	1			
⋮						
n-1					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
					↘	↓
n						$\binom{n}{k}$

Figure 1.3: Pascal's triangle

Again we have seen that memoization helps. With this in mind we can now turn out attention to some interesting problems in the following sections which have very nice Dynamic Programming algorithms. Not only these algorithms are excellent candidates for studying design and analysis of algorithms but also they help you better understand the notion of Dynamic Programming .

1.3 Subset Sum

1.4 Knapsack

1.5 Matrix Chain

1.6 A List of Problems

1.7 Another List

1.8 Summary

- We may apply Dynamic Programming to some optimization problem generally when Divide-and-Conquer and Greedy fails.
- We can only use Dynamic Programming whenever the problem exhibits both *optimal substructure* and *overlapping subproblem* properties.

APPENDIX A

Birth of 'Dynamic Programming'

> *An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.*
>
> Richard Bellman, *Eye of the Hurricane: An autobiography*, 1984

The Bellman-Ford shortest path algorithm presented earlier is named after Richard Bellman and Lester Ford. In fact that algorithm can be viewed as a dynamic program.

Although the quote is an interesting bit of history it does not tell us much about dynamic programming. But perhaps the quote will make you feel better about the fact that the term has little intuitive meaning! In short, the term programming essentially means planning (like in linear programming) while the word dynamic express the non static nature of the algorithm design.

Bibliography