Dynamic Programming

OWOMUGISHA ISAAC
15/U/12351/PS
215004131

Abstract

This paper is about dynamic programming, an important algorithm design technique. In this paper, I explore what dynamic programming is and show some examples of how it can be used to solve some problems efficiently. I'm writing this as a result of my fascination with the technique, which I first came across while studying recursion (and memoization) in my data structures and algorithms class. I decided to delve deeper and I have found that it is indeed quite a useful and beautiful algorithm design technique. In this paper, I present some thoughts on what I have learned.

1 Introduction

Dynamic programming is a general and powerful algorithm design technique. A large class of seemingly exponential problems have a polynomial solution only via dynamic programming.
Just like the divide and conquer method, dynamic programming solves problems by combining solutions to sub-problems, however, dynamic programming only applies when the sub-problems overlap, that is when sub-problems share sub-problems. When this happens, a divide-and- conquer algorithm would do more work than necessary. A dynamic programming programming algorithm solves each sub-problem and saves its answer in a table, thereby avoiding re-computing the answer every time it solves each sub-problem.
Dynamic programming is typically applied to optimization problems, that is, problems that might have many possible solutions but we are looking for the optimal solution.

2 The Technique

Dynamic programming can be thought of as a combination of "recursion" and "memoization". There are four steps involved in developing a dynamic programming algorithm:

- Characterize the structure of an optimal solution (this involves defining a base case and the sub-problems)
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution.
- Construct an optimal solution from computed information.

The following section illustrates how these steps apply to computing the n^{th} Fibonacci number, and the dynamic programming algorithm is compared with the naive recursive algorithm.

3 Dynamic Programming Example: Computing Fibonacci Numbers

Consider the Fibonacci numbers which are defined as follows:
$F_1 = F_2 = 1$
$F_n = F_{n-1} + F_{n-2}$ for $n > 2$.
We want an algorithm that takes as input an integer n, and returns as output F_n. Consider the following algorithm (implemented in python):

```
$def fib(n):$
  $if n <= 2 : f = 1$
  $else : f = fib(n-1) + fib(n-2)$
  $return\ f$
```

The runtime of this algorithm is $O(2^n)$. This is very inefficient. The major drawback of this algorithm is that it recomputes the same values many times. Dynamic programming eliminates the re-computation by storing values of $fib(k)$ in an array. The resulting algorithm runs in $O(n)$ time (quite a big improvement from the exponential algorithm). The algorithm (implemented in python) is shown below:

```
$memo = \{\}$# dictionary with keys, k and values = fib(k)
$def fib(n):$
  $if n$ in memo:$return\ memo[n]$
  $if n <= 2 : f = 1$
  $else : f = fib(n-1) + fib(n-2)$
  $memo[n] = f$
  $return\ f$
```

4 Conclusion

In this paper I have given a brief overview about dynamic programming and given an example of how it can improve an algorithm from exponential time to linear time. The topic of dynamic programming is wide and interesting. The reader is referred to ”Introduction To Algorithms by CLRS” for more on the topic.