Simplifying Dynamic Programming

Jamil Saquer and Lloyd Smith
Department of Computer Science
Missouri State University
Springfield, MO, USA
JamilSaquer/LloydSmith@missouristate.edu

***Abstract*— Dynamic Programming (DP) is one of the fundamental topics usually covered in the algorithms course. Nonetheless, it is a challenging subject for many computer science students. This paper presents work in progress for making dynamic programming easier for students. We propose four simple and interesting DP problems that students can learn early in the curriculum. One of these problems can be introduced as early as CS1. The other problems can be used in a CS2 course. We believe that this will make the subject of DP easier for many students and will result in more success and retention of CS students.**

Keywords—computer science education; dynamic programming

I. INTRODUCTION

Dynamic programming (DP) is an important design technique that students usually study in an algorithms course. It is used to solve optimization problems when a problem exhibits two properties: overlapping subproblems and optimal substructure [2]. Overlapping subproblems means that solving the original problem requires subproblems to be repeatedly solved again and again. A famous example is the Fibonacci sequence where computing Fibonacci(n) will require repeated calls of Fibonacci(m) for values of m less than n. A table is used to remember solutions of the smaller subproblems so the same subproblems do not need to be solved over and over again. This improves the running time of solving such problems to polynomial time. The optimal substructure property means that optimal solution of the original problem includes optimal solutions to subproblems. An example is finding the shortest path between two vertices u and v in a graph [1]. If the shortest path from u to v goes through vertex x then vertex y, then the shortest path from x to v must go through vertex y.

Mastery of DP requires good deal of practice from students [3]. To help students achieve this, we propose introducing DP early in the curriculum in a course such as CS2. We also propose starting with a set of simple DP problems which will help students better understand the technique of DP.

Recursion provides a good place for introducing DP to students. As a matter of fact, one can call DP recursion with "caching". Moreover, DP solutions to many problems start with recursive solutions. The recursive solution is then revised by caching solutions to smaller problems in a table to yield to a DP solution.

The Fibonacci sequence is an excellent problem to introduce DP to students. Since recursion is usually done in CS1 and the DP solution to the Fibonacci sequence is so easy, students can be introduced to DP as early as CS1.

The rest of this paper is organized as follows: sections II through V describe different simple DP problems that can be introduced early in the CS curriculum. Section VI discusses plans and ideas for future work.

II. THE FIBONACCI SEQUENCE

The Fibonacci sequence is the sequence 1, 1, 2, 3, 5, 8, 13 …etc. Formally, the i^{th} Fibonacci number is given by $F_i = F_{i-1} + F_{i-2}$, with $F_1 = F_2 = 1$. A straight forward recursive solution in Python is as follows:

```
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n – 1) + fibonacci(n – 2)
```

A DP solution is obtained by using a table to cache solutions to subproblems. There are two ways to implement a DP solution: top down and bottom up. In the top down approach, when a Fibonacci value is needed, we first look in the table for that value. Only if the value is not found in the table, the function will be called recursively and the value will be stored in the table before it is returned. As a result, the function won't be called more than once for smaller versions of itself. This is shown in the following O(n) implementation

```
d = {} # Dictionary for table to store values
def fibo(n):
    #Top down DP implementation
    if n == 1 or n == 2:
        return 1
    if d.get(n, None) == None: # if value is not stored
        d[n] = fibo(n – 1) + fibo(n – 2) # store it
    return d[n]
```

In the bottom up approach, a table is built starting with the basic case or cases of the recursive solution. Then the rest of the table is built in a bottom up fashion to reach the required solution. Usually, the solution of the problem is stored as the last entry of the table. This is shown in the following implementation:

```
table = [] # List/array for table to store results
def fibo(n):
    #Bottom up DP implementation
    table[1] = table[2] = 1
    for i in range(3, n + 1):
        table[i] = table[i – 1] + table[i – 2]
    return table[n]
```

On an Intel Core i7, 3.6 GHZ PC with 32 GB RAM, it took 8 milliseconds to compute fibo(500) on both top down and bottom up implementations. On the other hand, it took 30.4 seconds to calculate fibonnaci(40) using the naïve implementation --and it took 62.7 minutes to calculate fibonacci(50). Students in a CS1 course are usually astonished when they see these difference in performance. This is also a good example for students to appreciate the difference between different running times.

Some resources call the top down approach as memoization and the bottom up approach as Tabulation [3]. We won't be concerned with these names in this paper. However, the bottom up approach is usually easier for students to understand and we recommend using it when first introducing DP.

III. THE JUMP IT GAME

The "Jump It" game consists of a board with n cells or columns. Each cell, except the first, contains a positive integer representing the cost to visit that cell. The first cell, where the game starts, always contains zero. A sample game board where n is 6 is shown in Fig. 1.

We always start the game in the first column and have two types of moves. We can either move to the adjacent column or jump over the adjacent column to land two columns over. The cost of a game is the sum of the costs of the visited columns. The objective of the game is to reach the last cell with the cheapest cost.

In the board shown in Fig. 1, there are several ways to get to the end. Starting in the first column, the cost so far is 0. One could jump to 75, then jump to 43, and then move to 11 for a total of 75 + 43 + 11 = 129. However, a cheaper path would be to move to 5, jump to 7, then jump to 11, for a total cost of 5 + 7 + 11 = 23. We want to write a solution to this problem that computes the cheapest cost for a game board represented as an array. A straightforward recursive solution is as follows:

```
def jumpIt(board):
    #board – list representing playing board
    if len(board) == 1: #if board contains only one cell
        return board[0]
    elif len(board) == 2: #if board contains exactly two cells
        return board[0] + board[1] #must move to adjacent cell
    elif len(board) == 3: #if board contains exactly three cells
        return board[0] + board[2] #it is cheaper to jump over
    else:
        #lowest cost if next move is to move to adjacent cell
        cost1 = board[0] + jumpIt(board[1:])
        #lowest cost if next move is to jump over adjacent cell
        cost2 = board[0] + jumpIt(board[2:])
        return cost1 if cost1 < cost2 else cost2
```

0	5	75	7	43	11

Fig. 1. Game board with 6 cells.

To get a DP solution from the recursive solution, one should first write a recurrence equation that shows how the problem is solved in terms of smaller versions of itself. The recursive part in the jumpIt function gives us the equation we want

$$\text{jumpIt}(i) = \min\{\text{board}[i] + \text{jumpIt}(\text{board}[i+1:]),\ \text{board}[i] + \text{jumpIt}(\text{board}[i+2:])\}$$
$$= \text{board}[i] + \min\{\text{jumpIt}(\text{board}[i+1:]),\ \text{jumpIt}(\text{board}[i+2:])\}$$

This says that the lowest cost for playing the game starting at cell i is the cost for visiting cell i plus the minimum of the lowest costs of playing the game by moving to the adjacent cell or by jumping one cell over. Now, we can use a table to cache solutions to subproblems and follow a bottom up approach to reach a DP solution.

```
cost = [] #cache table to store solutions to subproblems
def jumpIt(board):
    #Dynamic programming implementation
    #board -- list with costs associated with visiting each cell
    n = len(board)
    cost[n −1] = board[n −1] #cost if starting at last cell
    cost[n −2] = board[n −2] + board[n −1] #cost if starting at
                                          #cell before last cell
    #now fill the rest of the table
    for i in range(n − 3, −1, −1):
        cost[i] = board[i] + min(cost[i + 1], cost[i + 2])
    return cost[0] #cost playing game starting at first cell
```

The cache table, cost, is a global list/array that stores solutions to subproblems. The actual size of this list is the same as the size of the game board. The value cost[i] gives the optimal cost of playing the game starting at index i. So, cost[0] is the optimal cost of playing the game starting at the first cell. Finding the actual optimal path taken from the first cell to the last can be done by reverse engineering the costs in the list cost.

IV. NUMBER OF WAYS TO CLIMB STAIRS

Assume that we have a stairway consisting of n steps. A child is standing at the bottom of the stairway and wants to reach the topmost step. The child can take 1, 2 or 3 steps at a time. What is the number of possible ways to climb the stairs [5]? To solve this problem, we first need to think about the special and simple cases. If n is 1, then there is only one possible way which is to take one step up. If n is 2, then there are 2 possible ways: take one step up followed by another or take two steps up. If n is 3, then the number of possible ways is 4 because the child can take one step followed by another followed by another, or take two steps up followed by one step, or take one step up followed by two, or take 3 steps up.

To write a recurrence formula that leads to a DP solution for this problem, we need to think about the general case, which is the possible ways to reach step i, where i is greater than 3. The last move in a path from the bottom of the stairway to the i^{th} step is one of the following:

1. From step i – 1, by taking one step up

2. From step i – 2, by taking two steps up

3. From step i – 3, by taking 3 steps up

Let us use the notation (m_1, m_2, …, m_w) to indicate the sequence of moves taken by the child in a path from the bottom of the stairway to a certain step in the stairway. Since the child can take 1, 2 or 3 steps at a time, m_i is 1, 2, or 3 for i = 1, 2, …, w. For example (2, 1, 2) indicates a 2-steps move, followed by a 1-step move, followed by a 2-step move. This sequence of moves will land the child on the fifth step. Table 1 shows the possible moves that can be taken to reach the first four steps in the stairway.

Notice that the possible paths to climb to the 4th step are done by adding 1 as the last move to a path to step 3, adding 2 as the last move in a path to step 2, or adding 3 as the last move to a path to step 1. Therefore, number of paths to step 4 is the sum of the number of paths to the previous three steps.

In general, if we let numWays(i) indicate the number of possible ways to reaching step i, then clearly

numWays(i) = numWays(i − 1) + numWays(i − 2) + numWays(i − 3)

It is easy to translate the above formula to a straightforward recursive function. Nonetheless, that solution will have an exponential running time. We notice that this problem is a good candidate for a DP solution because of repeated calls to subproblems. A possible implementation is as follows:

```
def numWays(n, table):
    '''DP implementation for the stairs climbing problem
       n -- number of steps in a stairway
       table -- a list to cache solutions to subproblems
       return number of possible ways to climb n steps
        precondition: n is at least 3
    '''
    table[1] = 1
    table[2] = 2
    table[3] = 4
    for i in range(4, n + 1):
        table[i] = table[i − 1] + table[i − 2] + table[i − 3]
    return table[n]
```

The running time for this implementation is linear compared to the exponential running time for a recursive implementation that does not use DP. If we restrict moves to only one or two steps up at a time, then this problem reduces to the Fibonacci sequence. The only difference will be numWays(2) = 2 while Fibonacci(2) is 1.

Step Number	**Possible Paths**	**Number of Paths**
1	(1)	1
2	(1, 1), (2)	2
3	(1, 1, 1), (2, 1), (1, 2), (3)	4
4	(1, 1, 1, 1), (2, 1, 1), (1, 2, 1), (3, 1), (1, 1, 2), (2, 2), (1, 3)	7

Table 1 - Different possible paths to climb the first four steps in a stairway.

V. ROBOT IN A GRID

Imagine a robot at the top left corner in a two dimensional grid. The robot can only move right or bottom to an adjacent cell. We want to find the number of possible ways for the robot to reach the bottom right cell [4]. Two possible paths in a 4 by 5 grid are shown in Fig. 2.

This problem can be approached similar to the previous one of finding number of ways to climb n steps. To approach this problem, let us ask how the robot can reach a given cell (r, c) in the grid.

Since there are two possible moves: right and down, cell (r, c) can be reached either from the cell to its left or from the cell above it. That is, cell (r, c) can be reached either from cell (r, c − 1) or from cell (r − 1, c). Therefore, the number of ways to reach cell (r, c) is the number of ways to reach cell (r, c − 1) plus the number of ways to reach cell (r − 1, c). The basic cases are when the cell is in the first row (r = 0) or in the first column (c = 0). If the cell is in the first row, the robot will keep moving right until it reaches cell (0, c). So, there is only 1 way to reach any cell in the first row. Similarly, if the cell is in the first column, the robot will keep moving down until it reaches the cell (r, 0). This gives the basis for a recursive solution. However, a naïve implementation will result in an exponential running time because the code will be repeatedly called for the same location. We can avoid this by using a DP approach where we cache results. This leads us to the following solution:

```
def numWays(table):
    '''DP implementation for the robot in a grid problem
       table -- 2D list to store results where table[r][c] stores
       number of ways to reach cell (r, c) in a 2D grid.
       table has the same number of elements in each
       dimension as the grid
    '''
    numRows = len(table)
    numColumns = len(table[0])

    #fill entries in the first column
    for row in range(numRows):  table[row][0] = 1
    #fill entries in the first row
    for column in range(0, numColumns): table[0][column] = 1

    #fill the rest of the table
    for row in range(1, numRows):
        for column in range(1, numColumns):
            table[row][column] = table[row - 1][column] +
                                 table[row][column - 1]
```

The above DP implementation has quadratic running time. Table 2 shows the cache table corresponding to the grid from Fig. 2. The number of ways to reach the bottom right cell is stored in the last (i.e., bottom right) cell in the table.

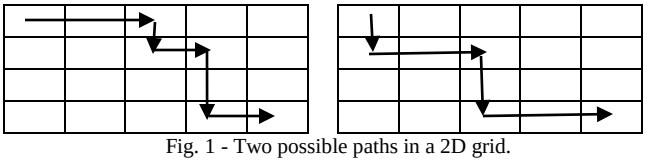

Fig. 1 - Two possible paths in a 2D grid.

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35

Table 2 - Number of possible ways to reach the different cells in a 4 by 5 grid.

An interesting and easy generalization to the above problem is to consider a grid where some of the cells have obstacles and therefore cannot be visited. If a cell (r, c) has an obstacle, then number of ways to reach that cell is 0 and we store the value 0 at table[r][c]. Another interesting generalization is to find an actual path for the robot to follow from the top left cell to the bottom right cell in the presence of obstacles. This is done by starting at the last cell and going backwards to follow cells that lead to the current cell. A path is found when cell (0, 0) is reached. It is possible for some grids with obstacle that a path to the last cell may not exist.

VI. DISCUSSION AND PLANS FOR FUTURE WORK

One of the authors introduced the concept of caching solutions and the notion of DP to students in CS1 while discussing recursion. Similar thing was done in CS2 but it was done in the context of discussing big-O notation and different implementations of the Fibonacci sequence. Both groups of students liked the technique especially when different implementations of the Fibonacci sequence were demonstrated in class.

We plan to introduce DP in CS2 in the fall semester using the problems presented in Sections II through V. Data will be collected to assess students' understanding of the topic. We will also collect data from two groups of students in the algorithms class. One group would have been introduced to DP while the other group would have no prior exposure to DP. Comparison between the two groups will be done. We expect that prior exposure to DP in CS2 to help students' understanding of the topic in CS2.

REFERENCES

[1] T. Cormen, Algorithms Unlocked, Cambridge, MA: MIT Press, 2013.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., Cambridge, MA: MIT Press, 2009.

[3] J. Kleinberg and A. Tardos, Algorithm Design. Boston, MA: Pearson, 2005.

[4] G. McDowell, Cracking the Coding Interview: 150 Programming Questions and Solutions, 5th ed., Palo Alto, CA: CareerCup, 2011.

[5] Tutorial Horizon, http://algorithms.tutorialhorizon.com/dynamic-programming-stairs-climbing-puzzle/, accessed 10/9/2016.