An Approach to Making Dynamic Programming Easier for Students in the Computer Science Curriculum

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

ABSTRACT

Dynamic Programming (DP) is an important subject in the computer science curriculum that is usually covered in an algorithms course. Nonetheless, many students find DP difficult to understand. This paper presents an approach to making the subject easier for students. Our approach is based on introducing the topic early in the curriculum and on starting with easy and interesting problems. We report on our experience applying this approach to students in a CS2 class, which showed positive results.

KEYWORDS


1 INTRODUCTION

Dynamic programming (DP) is an important programming technique that students usually study in an algorithms course. It is used to solve optimization problems where a problem is decomposed into smaller sub-problems or versions of itself. Solutions to the sub-problems are then used to reach a solution to the original problem. A problem is usually considered a good candidate for dynamic programming if it exhibits two characteristics: overlapping sub-problems and optimal substructure [1]. Overlapping sub-problems means that solving the original problem requires sub-problems to be repeatedly solved again and again. A famous example is the Fibonacci sequence where calling Fibonacci(n) to find the n\textsuperscript{th} Fibonacci number requires repeated calls of the Fibonacci function for finding smaller Fibonacci numbers. Moreover, the function is called multiple times for the same argument. For example, finding Fibonacci(5) requires Fibonacci(3) to be called twice and Fibonacci(2) to be called three times as can be seen in the following recursion tree.

In DP, a table is used to remember solutions of the smaller sub-problems so that the same sub-problems 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 sub-problems. An example is finding the shortest path between two vertices u and v in a graph [2]. 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. An illustration of this is shown in Figure 2.

![Figure 2. Shortest path from u to v goes through x and y. So shortest path from x to v must go through y. Numbers represent lengths of paths.](image)

Mastery of DP requires a great deal of practice from students [3]. To help students achieve this, we propose introducing DP early in the CS curriculum in a course such as CS2. We also propose starting with a set of simple and interesting problems to help students better understand the technique of DP. We expect that this approach will make the topic easier for students when they encounter it in a later course such as the algorithms course, which is where students usually study DP in details.

Recursion provides a good way to introduce 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 a DP solution.

The Fibonacci sequence is an excellent problem for introducing 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 2 through 5 describe different simple DP problems that can be introduced early in the CS curriculum. We describe our experience and assessment of introducing DP to students in a CS2 course in Sections 6 and 7. Conclusions and plans for future work are given in Section 8.

## 2 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 straightforward recursive solution in Python is as follows:

```python
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.

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

```python
table = [] # List to store results

# Bottom-up DP implementation
for i in range(3, n + 1):
    table[i] = table[i -1] + \n    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 fibonacci(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 such difference in performance. This is also a good example for students to appreciate the difference between different running times.

Some resources refer to 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.

### 3 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 Figure 3.

![Figure 3. Game board with 6 cells.](image)

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 lowest cost.

In the board shown in Figure 3, there are several ways to get to the end. Starting in the first column, the cost so far is 0. One could jump to 4, then jump to 5, and then move to 7 for a total of 4 + 5 + 7 = 16. However, a lower-cost path would be to jump to 4, move to 3, and then jump to 7, for a total cost of 4 + 3 + 7 = 14. We want to write a solution to this problem that computes the cheapest cost for a game board represented as an array.

It is usually a good idea to start with a recursive solution and use that as a guide for the DP solution. To build a recursive solution, we need to think about the special case or cases. If the board has only one cell, then the cost of playing the game is the cost of visiting that cell. This case can be interpreted as playing the board starting at the last cell on the board. Another basic case is when the board has exactly two cells. This can be interpreted as playing the game starting with the cell just before the last cell. In this case, it would be cheaper to jump over the adjacent cell. So, the cost of playing the game is the cost of visiting the first cell plus the cost of visiting the last cell. In this case, the cost of playing the game is the cost of visiting these two cells. A third basic case is if the board consists of three cells, which can be interpreted as starting the game two cells off the last cell. In this case, the cost of jumping the game is the cost of the adjacent cell plus the cost of playing the game if the next move is into the adjacent cell with the cost of playing the game if the next move is to jump over the adjacent cell. Then the total cost of playing the game is the cost of visiting the current cell plus the minimum of the above two costs. A recursive solution that uses these cases is as follows:

1 Python’s lists are similar to arrays in other programming languages such as Java and C++
def jumpIt(board):
    #board - list representing playing board
    if len(board) == 1:
        #if board contains exactly 1 cell
        return board[0]
    elif len(board) == 2:
        #if board contains exactly 2 cells
        #then must move to adjacent cell
        return board[0] + board[1]
    elif len(board) == 3:
        #if board contains exactly three cells, it is cheaper to jump over
        return board[0] + board[2]
    else:
        #find lowest cost if next move is to adjacent cell
        cost1 = board[0] + \
            jumpIt(board[1:])
        #find lowest cost if next move is to jump over adjacent cell
        cost2 = board[0] + \
            jumpIt(board[2:])
        if cost1 < cost2:
            return cost1
        else:
            return cost2

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:

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

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 sub-problems and follow a bottom-up approach to reach a DP solution as is shown in the following Python implementation.

cost = [] #dictionary/table for caching solutions to sub-problems
def topDownJumpIt(board, i):
    n = len(board)
    #If value is not in cache table
    if cost.get(i, None) == None:
        #calculate value and cache it
        if i == n - 1:
            cost[i] = board[n - 1]
        elif i == n - 2:
            cost[i] = board[n - 2] + \n                board[n - 1]
        else: #board length is >= 3
            #find cost if next move is into adjacent cell
            cost1 = \n                topDownJumpIt(board, i+1)
            #find cost if next move is jump over adjacent cell
            cost2 = \n                topDownJumpIt(board, i+2)
            #board- list with costs associated
            #with visiting each cell
            if cost1 < cost2:
                cost[i] = cost1
            else:
                cost[i] = cost2

The cache table, cost, is a global list/array that stores solutions to sub-problems. 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.

A top-down DP solution to the Jump-It problem is not hard and is shown below as the Python function topDownJumpIt. The function takes as parameters a list, board, representing the playing board with costs of visiting the different cells and an integer, \(i\), representing the index on board of the cell where the game starts. The function returns the minimum cost of playing game starting at cell \(i\).
Finding the actual optimal path taken from the first cell to the last can be done by using an additional list, say path, where path[i] stores the index of the cell visited after cell i. Whenever a decision is made regarding whether the cheapest cost of playing the game starting at cell i is by moving into an adjacent cell (i.e., cell i + 1) or by a jump over (i.e., cell i + 2), then path[i] is updated to store the index of the next cell on the path.

### 4 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 [4]? 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 on 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 \((m_1, m_2, \ldots, m_w)\) to indicate the sequence of moves taken on 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 \(i = 1, 2, \ldots, w\). 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.

<table>
<thead>
<tr>
<th>Step Number</th>
<th>Possible Paths</th>
<th>Number of Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>(1, 1), (2)</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>(1, 1, 1), (2, 1), (1, 2), (3)</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>(1, 1, 1, 1), (2, 1, 1), (1, 2, 1), (3, 1), (1, 1, 2), (2, 2), (1, 3)</td>
<td>7</td>
</tr>
</tbody>
</table>

Notice that the possible paths to climb to the 4\(^{th}\) 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 \(\text{numWays}(i)\) indicate the number of possible ways to reach step i, then clearly

\[
\text{numWays}(i) = \text{numWays}(i - 1) + \text{numWays}(i - 2) + \text{numWays}(i - 3)
\]

It is easy to translate the above formula to a straightforward recursive function. Nonetheless, this 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 sub-problems. A possible implementation is as follows:

```python
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 sub-problems
    return number of possible ways to climb n steps
    precondition: n is at least 3
    '''
```
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 is that numWays(2) = 2 while Fibonacci(2) is 1.

5 ROBOT IN A GRID

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

![Figure 4. Two possible paths in a 2D grid.](image)

This problem can be approached in a similar way to the previous problem of finding the 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 or 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 one 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:

```python
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]
    return table[n]
```

The above DP implementation has quadratic running time. Table 2 shows the cache table corresponding to the grid from Figure 4. The number of ways to reach the bottom-right cell is stored in the last (i.e., bottom-right) cell in the table.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>10</td>
<td>20</td>
<td>35</td>
</tr>
</tbody>
</table>

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 \( \text{table}[r][c] \). 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 can be 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 grids with obstacles that a path to the last cell may not exist.

### 6 EXPERIENCE INTRODUCING DP IN A CS2 COURSE

DP was introduced to students in a CS2 course in the fall 2016 semester. Two 50-minutes lectures explained the technique using the problems discussed in the previous sections. The goal was to have students recognize the two properties of overlapping sub-problems and optimal substructure that a problem should have to be a good candidate for solution using a DP approach. Students were also introduced to both the top-down and bottom-up approaches of DP. The first two examples (Fibonacci numbers and Jump-It game) were explained using both the top-down and bottom-up approaches. The last two examples (number of ways to climb stairs and robot in a grid) were explained using only the bottom-up approach. More emphasis was given to the bottom-up approach because we felt that it was easier for students to understand. Using the first two examples, students were also shown how an inefficient recursive solution could be converted to an efficient solution using DP.

To assess students’ understanding of the topic and have them practice the technique, two 100-minute labs, one homework assignment and a short multiple-choice quiz were given. The first lab was given in-between the two lectures and its goal was to familiarize students with DP. For this lab, students were asked to provide both a top-down and bottom-up implementations of a familiar problem, which is finding the \( n^{th} \) Fibonacci number. The second lab and the homework assignment are discussed in the following subsections.

#### 6.1 Lab - Minimum Cost Path

The goal of this lab was to have students recognize the two characteristics of overlapping sub-problems and optimal substructures and to provide a DP implementation of an interesting problem [6]. The statement of the problem is as follows. Assume that you are given a two-dimensional cost matrix, where each cell contains a cost associated with visiting that cell. A sample matrix is shown in Figure 5.

![Figure 5. Cost matrix](image)

We want to find the lowest cost of a path that leads from cell \((0, 0)\) to any cell \((m, n)\) in the matrix. The cost of the path is the sum of the costs of all the cells on that path including the source and destination cells. You can assume that all costs are positive. You are allowed to travel from cell \((0, 0)\) to cell \((m, n)\) by either moving right, down, or diagonally to an adjacent cell in the matrix. That is, from a given cell \((i, j)\), you can move to any of the cells \((i, j + 1), (i + 1, j), \) or \((i + 1, j + 1)\), which represent movements to the right, down and diagonally, respectively. For example, Figure 6 shows the minimum cost path from cell \((0, 0)\) to cell \((2, 3)\). The path follows cells \((0, 0) \rightarrow (0, 1) \rightarrow (0, 2) \rightarrow (1, 3) \rightarrow (2, 3)\) and the cost of the path is \(3 + 2 + 1 + 2 + 3 = 11\).

![Figure 6. Arrows show the minimum cost path from cell (0, 0) to cell (2, 3)](image)
three cells (m-1, n-1), (m-1, n), or (m, n-1). So a solution to the problem for cell \( (m, n) \) is based on – or contains within it – knowing the solutions for cells (m-1, n-1), (m-1, n), and (m, n-1). Therefore, the minimum cost to reach cell \( (m, n) \) can be written as the minimum of the minimum costs of reaching these 3 cells plus the cost of cell \( (m, n) \). This can be written as the following recurrence relation, where \( \text{minCost}(m, n) \) is the minimum cost of a path from cell \( (0, 0) \) to cell \( (m, n) \).

\[
\text{minCost}(m, n) = \text{cost}[m][n] + \\
\min\{\text{minCost}(m-1, n), \\
\text{minCost}(m, n-1), \\
\text{minCost}(m-1, n-1)\}
\]

It is clear from the recurrence relation that this problem satisfies the overlapping sub-problems property.

Figure 7 shows the costs of the minimum cost paths from cell \( (0, 0) \) to the different cells in Figure 5. The content of cell \( (m, n) \) is the cost of the minimum cost path to that cell.

\[
\begin{array}{cccc}
3 & 5 & 6 & 10 \\
4 & 7 & 10 & 8 \\
9 & 5 & 9 & 11
\end{array}
\]

Figure 7. Minimum cost paths from cell \( (0, 0) \) to the different cells in Figure 5.

An interesting generalization of this problem is to find the actual minimum cost path in addition to finding the cost of that path. In the spring 2017 semester, we plan to show the students in the CS2 course how to find the actual path for the Jump-It problem. We expect students to be able to follow a similar approach for finding the minimum cost path for the lab problem.

6.2 Homework Assignment – Maximum Number of Golden Coins

An optimization problem was given as a homework assignment to reinforce the material learned in class and the lab. A brief statement of the problem is as follows. Given an \( m \) by \( n \) table, where each cell contains a certain number of golden coins. Find the maximum number of coins one can collect on a path from the top-left cell to the bottom-right cell. At each step, one can move one step right or down into an adjacent cell. For example, the maximum number of coins that can be collected using the data in Table 3 is 20.

Table 3. Number of golden coins in each cell of a table.

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

A cache table that shows the maximum number of coins that can be collected on a path from the top-left corner to any cell \( (m, n) \) in Table 3 is shown in Table 4.

Table 4. Cell \( (m, n) \) contains the maximum number of coins that can be collected on a path from the top-left cell to any cell \( (m, n) \).

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>8</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

7 RESULTS AND DISCUSSION

Table 2 gives a summary of students’ grades on the labs, homework assignment and quiz given over the DP material. The row labeled "Adjusted Average" is computed by removing some 6-7 zeros (out of 48 students) that correspond to students that did not complete the assignment.

Table 5. Dynamic programming assessment results.

<table>
<thead>
<tr>
<th></th>
<th>Lab 1</th>
<th>Lab 2</th>
<th>HW</th>
<th>Quiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Minimum</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Minimum Removing 0s</td>
<td>63%</td>
<td>95%</td>
<td>80%</td>
<td>25%</td>
</tr>
<tr>
<td>Median</td>
<td>100%</td>
<td>97%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Mean</td>
<td>87%</td>
<td>83%</td>
<td>77%</td>
<td>76%</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>0.31</td>
<td>0.35</td>
<td>0.40</td>
<td>0.35</td>
</tr>
<tr>
<td>Adjusted Average</td>
<td>97%</td>
<td>98%</td>
<td>98%</td>
<td>87%</td>
</tr>
</tbody>
</table>
The students in this course conduct their labs in a peer-programming format. An instructor and teaching assistant supervise students during the lab. Homework assignments and quizzes are individual efforts. The high grades reflect students’ understanding of the material. Moreover, the students found this material exciting. There were two outcomes they found appealing:

1. The runtime improvements provided by the DP techniques.
2. The relative ease in implementing these changes, i.e., in going from a naïve recursive solution to an elegant solution based on DP principles.

8 CONCLUSIONS AND PLANS FOR FUTURE WORK

This paper introduced an approach to making dynamic programming easier for students in the computer science curriculum. We presented four interesting problems with easy DP solutions. These problems were used by the authors to teach the topic to students in a CS2 course. Our assessments indicate that the students learned the topic successfully. In future semesters, we plan to increase our coverage of DP in CS2 by showing students how to extend a DP solution to find the actual choices that lead to the optimal solution found by DP.

As a follow up study, we plan to assess the impact of our approach on students’ understanding of DP in an algorithms course where harder problems are considered. Data will be collected from two groups of students. 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 the algorithms course.

REFERENCES