ABSTRACT
Many students have difficulty understanding the concept of dynamic programming, a problem solving approach appropriate to use when a problem can be broken down into overlapping sub-problems. In addition, computer science majors seldom take a course in which a spreadsheet application is taught or used, although spreadsheet applications can be a great tool in visualizing and solving problems. This paper attempts to address these issues by providing examples of problems that can be solved in a spreadsheet environment using dynamic programming in various computer science courses.

INTRODUCTION
There is a well-known approach to solving certain algorithms called Dynamic Programming (DP), an unusual term that resulted from the fact that the 1950 Secretary of Defense hated mathematics [2]. It is primarily a bottom-up method that has been around since the 1940’s used for developing a recursive, top-down solution for many problems. It is appropriate for use with problems that can be solved by breaking the problem down to overlapping sub-problems. The popularity of this approach is heavily due to its efficiency. A solution having an exponential complexity for a top-down recursive algorithm may have a polynomial complexity with the associated dynamic programming method. When students are first introduced to this approach as a problem solving method, they are often uncomfortable or at least have trouble visualizing the process. The teaching method discussed in this paper attempts to address these educational road-bumps. A collection of problems from very simple to somewhat complicated are solved in a spreadsheet environment using the dynamic programming
approach to demonstrate its algorithm visualization value. A positive side effect is that computer science majors become better acquainted with spreadsheet programming.

BACKGROUND

There has been previous work in combining MS Excel and Dynamic Programming (DP) particularly in the area of operations research modeling. Jensen’s website Operations Research Models and Methods is a useful site from which to download Excel add-ins that solves a variety of problems using dynamic programming. This software is in support of his book of the same name [4]. Professor Mark Gerstein in his Yale bioinformatics lab has been using similar techniques for visualizations of DNA sequence matching (basically a variation of the longest common subsequence problem) as well as other problems [3]. Consequently many of the techniques that will be discussed may well have been used in a variety of educational settings although it doesn’t appear to be widespread in dynamic programming. It is the purpose of this paper to demonstrate educational applications of Excel to DP. It is not the purpose of this paper to explain the dynamic programming algorithmic approach.

SIMPLE EXAMPLES

Algorithm designers generally consider problems suitable for dynamic programming if the problem can be modeled by top-down recursive representations of sub-problems, which have an optimal substructure. The efficiency of the approach is greatly increased if the sub-problems have a large amount of overlap. This will become clear within the context of some of the examples.

We begin with some very simple examples that many may not even consider to be DP due to their simplicity. Skipping probably the simplest, which is the calculation of factorial(n), we will start with the calculation of the nth Fibonacci number. Many computer science students have written this function recursively as one of the traditional problems presented when introducing the concept of recursion. Due to the exponential complexity one would not usually use the algorithm shown in Figure 1 so we turn out attention to the algorithm in Figure 2. This algorithm is a bottom-up solution to the top-down recursion solution shown in Figure 1. In other words, it is DP in a very simple form. The resulting complexity is O(n) not O(c^n), a significant improvement.

```c
int Fib( int n)
{
    if (n<3) return 1
    else return ( Fib(n-1)+Fib(n-2) );
}
```

Figure 1: Recursive Fibonacci.

```c
int Fib( int n)
{
    A=new int[n+1];
    for(int i=3 ; i<=n ; i++)
    return A[n];
}
```

Figure 2: Iterative Fibonacci.
Switching from C++ to a spreadsheet environment (MS Excel, in this case) one can quickly build a simple visualization of the Fibonacci process. Place the value 1 in cells A1 and B1 and the formula A1+B1 in cell C1. Copying C1 to the cells to the right immediately constructs the Fibonacci sequence in a bottom-up DP manner. This is, of course, a very simple example in which the value in each cell is determined by the sum of the previous two cells.

Moving from this one-dimensional setting to two-dimensional provides a more interesting scenario. Suppose we want to determine a particular coefficient of the binary expansion, i.e. \((a+b)^n\). The solution is a cell in Pascal’s Triangle. The \(n\)th row, \(k\)th column of this triangle is given by the recursive definition

\[
Pascal(n,k) = Pascal(n-1,k) + Pascal(n-1,k-1)\]

\[
Pascal(n,0) = Pascal(n,n) = 1
\]

Since writing a program that returns the value of a cell in the triangle recursively is computationally exponential, other approaches are often used. As with the Fibonacci array we can evaluate a cell by filling required portions of the array bottom-up. This approach, which is the usual procedure an instructor applies when the triangle is written on the board, is another dynamic programming example. If we can develop the triangle in a spreadsheet, one can see the triangle and its values easily. Enter the value 1 in the cell A1 and also in the row 1 and column A of the triangle. (See Figure 3.) Next, place the formula \(=A2+B1\) in cell B2 and copy to the remaining cells; this builds the triangle. Although it’s too fast to see as it develops, the cells are filled from the upper left to lower right. The evaluation looks to be dynamic programming even if the software within Excel works differently. The main point is that it takes very little Excel programming, in this case at least, to build and evaluate a dynamic programming table.

MORE COMPLEX EXAMPLES

Due to the simplicity of the previous problems, the dynamic programming process might not be clear to those unfamiliar with the process. To remedy this, the well-known Shortest Common Subsequence problem will be addressed [1]. Although more complex variations of this problem are heavily used in bioinformatics and DNA string matching, we use the simplest version for this paper.

A subsequence of a sequence \(X\) (think string, if you must) is a sequence obtained from \(X\) by removing 0 or more of the characters of \(X\) with the remaining characters kept in the same order. For example the sequence “abrdfrefdl” is a subsequence of \(X = \)
“abfrdfdveddfgfd”. The subsequence is obtained by removing the bold letters of X. Define a *Common Subsequence* of sequences X and Y as any sequence that is a subsequence of both X and Y. There may be many subsequences of X and Y. A subsequence of this set that has maximal length is called the *Longest Common Subsequence* (LCS), and there may be more than one of these. The *Common Subsequence Problem* is one in which we seek to determine a least one of the maximal sequences for two given sequence X and Y. To solve this problem requires some definitions, an approach that common in dynamic programming. Let Xi represent the first i characters of X and Yi the first i characters of Y. Assuming that X_n = x_1x_2x_3…x_n and Y_n = y_1y_2y_3…y_n and let Z = z_1z_2z_3…z_k be any LCS of X and Y. Now there are three situations that require consideration. Note LCS is defined recursively in terms of smaller sub-problems.

1. If x_k = y_k then z_k = x_k = y_k and Z_k is the LCS of X_{m-1} and Y_{n-1}.
2. If x_k ≠ y_k then z_k = x_k implies that Z is an LCS of X_{m-1} and Y.
3. If x_k ≠ y_k then z_k = y_k implies that Z is an LCS of X and Y_{n-1}.

Combining these in a recursive definition gives:

\[
c[i,j] = \begin{cases} 
0 & \text{if } x = 0 \text{ or } j = 0 \\
c[i - 1,j - 1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\
\max(c[i,j - 1], c[i - 1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j
\end{cases}
\]

where c[i,j] is defined to be the length of the LCS of prefixes X_i and Y_j. Note, the length of the LCS is defined, not the actual sequence.

The above definition specifies the value of the cells in a 2-dimensional table. For example, placing two strings, e.g. BDCABA and ABCBDAB in the indicated row and column respectively, the initial table is built as shown in Figure 4. This table is filled with the two strings and the initial 0 values in row 3 and column C. The cells in the D4:I10 rectangular area represent the values for c[i,j]. In particular c[1,1] is stored in cell D4 and c[2,3] is stored in F5 and so forth. To evaluate this table enter an Excel function for cell D4 and copy to the remaining cells. This function is the previously defined formula for c[i,j]; that is, IF(DS2=$B4,C3+1,MAX(D3,C4)). (It is important to note that many computer science students have little experience with Excel. They occasionaly build tables and charts for papers and such, but seldom take a course that trains one in MS Office tools. Thus, a simple review of absolute addressing and functions such as IF is probably called for.) Placing this formula in cell D4 and copying to the remaining cells generates the table on the right of Figure 4. From the lower right hand cell in the table one can see that the length of the longest common subsequence is 4. The length of the LCS is increased by 1 every time two of the characters from the strings match. Note that cell F6 is obtained by incrementing E5 by 1, a result of the fact that both F2 and B6 contain a C. The sequence of characters that generate this behavior is the LSC, i.e. BCBA in this case.
As a final example we show that it is possible to push this concept too far. There are problems for which a DP solution is complex enough that retrofitting the algorithm within Excel becomes overly complicated. An example of this is the well-known Optimal Binary Search Tree problem [1]. The problem is to construct a binary search tree in such a way as to optimize search time, given the probability of searching for each key. In other words, the tree should produce the minimal expected search cost. In the interest of brevity, let us consider the DP recurrence relation that results from this problem and omit the reasoning process that generated the relation. Solving this problem requires the constructions four tables (tables) e, weight (w), root and probability (Figure 5). The cells c(i,j) in the e array contains the cost of searching the sub-trees containing nodes i to j. The weight array contains the sums of the probabilities of the indicated nodes. The final answer is contained in the root array which indicates the structure of the optimal tree.

The recurrence for e is

\[
e[i,j] = \begin{cases} 
\min_{r \in \mathbb{R}} \{ e[i, r - 1] + e[r + 1, j] + w(i, j) \} & \text{if } j = i - 1 \\
i \leq j 
\end{cases}
\]

where

\[
w(i, j) = \sum_{x=i}^{j} p_x + \sum_{x=1}^{j} q_x
\]

From the recurrence relation, the tables shown in Figure 5 are built. The w array and the p,q array, which contains the initial set of probabilities of accessing the nodes, are easily established leaving the e array and the root array to be calculated. DP is applied to the e array and during the process the root array is filled.

To clarify the problem, first note that the e array is constructed from the lower right to upper left with the last step being the determination of cell(5,1) which is 2.75. Each cell is evaluated by used the previously calculated values below and to the right of
the cell as well as using the corresponding weight cell from the w array. For example cell(6,2) which is 1.2 can be determined as follows. (Figure 6.)

\[
\text{Cell}(6,2) = \min \{ \text{cell}(7,2) + \text{cell}(6,5), \text{cell}(8,2) + \text{cell}(6,4), \text{cell}(9,2) + \text{cell}(6,3) \} + .5
\]

(from the w array)
\[
\text{Cell}(6,2) = \min \{ .7+.05, .4+.3, .1+.6 \} + .5 = 1.2
\]

This process is straightforward to write in C++, but can we do it in Excel? It is not as easy as one might think. Developing an Excel formula for the cells that calculate the formula required in cell(6,2) is not obvious. Cell(5,1) requires a minimum applied to four sums, the next diagonal of numbers require 3 sums and so on. One could write each of these and copy diagonally but this is time consuming and the resulting DP visualization becomes confusing and less satisfying to a student. One reasonable solution is to write an Excel user defined function (UDF) using Visual Basic (VBA). VBA is the built-in macro language used by Excel. To investigate the effort required to write a UDF, we wrote the function in Figure 7. This function can then be copied to each cell and the values will be calculated as desired. However this method causes additional problems. Most students and perhaps the faculty as well, have little advanced knowledge of Excel. (That is true in our department.) Spending time learning the system and methods for building VBA functions in Excel might be considered getting off-track of the goals of this project. There is also another problem within Excel itself. Within the function definitions, code was included to modify the root array as it ran. The values in the root array result from the minimum sums in the e array. This is not possible as Excel does not allow cell modification side effects within functions. Alternatively, one could write a separate function to be used in the root array table but this would just be a
modified copy of minbin(). What’s the solution? An obvious method that would work efficiently is to write the entire algorithm as a procedural macro in VBA and have it fill in all the cells. Of course, in this case why not just use C++ in the first place? The moral of the story is that some DP problems are just too complicated to easily implement in Excel, especially if the original reason for doing so was a better understanding and visualization of dynamic programming tables.

CONCLUSION

Over the years it has become clear to the authors that tools such as Excel (or other spreadsheet software) are not appropriately appreciated by many computer science students. Most never take a course in Excel and hence are quite unaware of its advanced features beyond simple applications.

We make a conscious attempt to embed Excel into many of our courses so that the students will at least be exposed to spreadsheets concepts. In addition, it is a good tool to visualize table construction algorithms. In particular, it has been used in Data Structures to graph data, in Introduction to Computing for Science Majors to build tables such as Fibonacci and Pascal and in Bioinformatics in which the Longest Common Subsequence Algorithm is always investigated. In other words, since most computer science programs do not include a course in spreadsheets, it is desirable to include it throughout a computer science program. Dynamic programming techniques with their associated visualization are one of the most interesting, as well as educational applications, of a spreadsheet in a computer science program.

REFERENCES

