Dynamic Programming Algorithms : Design, Analysis and Examples
Adapted from Russell Impagliazzo

When is dynamic programming useful Dynamic programming is useful whenever you have a recursive procedure that is making *repeated* calls on *equivalent* sub-problems. Since these repeated calls themselves call the recursive procedure several times, this can lead to an unnecessary exponential blow-up in total time. In dynamic programming, we identify these repetitions, and *store and reuse* the answers, instead of repeating them from scratch. Normally, we then rewrite the program into an *iterative* algorithm that solves the sub-problems in *bottom-up* order. Dynamic programming typically is used in *search and optimization* problems.

Basic Design Steps 1. Find a recursive solution for the problem. Usually, a simple one will be more likely to have repeated sub-problems than a complicated one. Generally, you should try to minimize the number of parameters that need to be carried forwards in the procedure.

2. Identify and classify repeated sub-problems. What kind of sub-problems does your algorithm use? Why might there be equivalent sub-problems? What do the problems at level i of the recursion look like? Think of a way to label each sub-problem that might come up. The labels should be tuples of small integers, keeping track of different parts of the sub-problem.
3. Use the labeling scheme to come up with the dimensions of the table to store answers to sub-problems.
4. Identify what is the "top-down" order. What changes in a sub-problem called by a procedure as opposed to the input problem? Then invert this order, to define a "bottom-up" order for sub-problems.
5. Now you have all the ingredients for your DP algorithm. The outline will go something like:
   Generic DP:
   1 Initialize array $Answer[possible sub-problem label]$ to store the answers for sub-problems.
   2 Fill in the values of the "base cases" of the recursion. For each (sub-problem where the recursion bottoms out) do: (fill in $Answer[sub-problem]$ according to base case of recursion)
   3 In bottom-up order, fill in all other sub-problems according to recursion: For (first sub-problems in bottom-up order) to (last sub-problem in bottom up order) do: $Answer[sub-problem] \leftarrow$ Recursive body changing indices as needed and replacing recursive calls on sub-problem' with $Answer[sub-problem']$.
   4 Return $Answer[main problem]$.
6. Correctness follows from the correctness of the recursion, since we do the same work as the recursion, just in different order and without repetitions. Time analysis is based on the new loop structure. Basically, it is $Time$ = # of sub-problems $\times$ Time to do main body of loops. Another way to think of this is the $Time$ = cost of filling up each element of the table $\times$ # of elements

rather we do something more along the line with the format of Quiz 2: First, we give a definition of the subproblem; next we list the base case; then we give the recurrence relation for the problem; finally we describe the form that the table will take. As practice try and fill in the above the steps that aren't specifically mentioned in the examples.

1 Pascal's Triangle

Pascal's Triangle is a "triangle" of numbers where each element is the sum of the two elements above, except for the edge elements which are one:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 . . . . . .
```

But to put it into a form amenable to Dynamic Programming we turn it on it's side:

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

Problem Definition $PASCAL(r, c)$ is the $c+1$th element in the $r+1$th row of Pascal's triangle.

Base Case $PASCAL(r, 1) = PASCAL(c, 1) = 1$.

Recursive Solution (including base case)

$$PASACL(r, c) = \begin{cases} 1 & c = 1 \text{ or } r = 1 \\ PASCAL(r, c-1) + PASCAL(r-1, c) & \text{otherwise} \end{cases}$$

Table Description PASCAL is an $n \times n$ 0-indexed table. Once the base cases are added, it can either be filled row by row or column by column.

2 MCP

[CLRS 331-9] We are given a sequence $< A_1, A_2, \ldots, A_n >$ of n matrices to be multiplied, and we wish to compute the product $A_1 A_2 \cdots A_n$. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix product. All parenthesizations yield the same product, but the parenthesization that we choose can have a dramatic affect on the cost of evaluating the product. If B is a $p \times q$ matrix and C is a $q \times r$ matrix the resulting matrix D is a $p \times r$ matrix. The cost of computing this product is dominated by the total number of scalar multiplications which is pqr. Depending on how the matrix products are parenthesized, we can limit the dimensions of the matrices involved in the product and thus save computational cost. The MCP problem is that of finding the parenthesization of the product of matrices which minimizes the number of scalar multiplications required.

Problem Description $MCP(i, j)$ is the minimum amount of multiplications achievable by parenthesizing the sub-chain from matrix i to matrix j.

Recursive Solution (including base case)

$$MCP(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j}(MCP(i,k) + MCP(k+1,j) + p_{i-1}p_k p_j) & \text{if } i < j \end{cases}$$

Table Description MCP is a 0-indexed $n \times n$ table. Where its row index corresponds to the start point of the chain, and its column index corresponds to the ending point of the chain. It is filled in order of increasing chain length, so first the cost of all chains of length two are computed, then length three, ..., chains of length n. This leads to the half of the matrix of above the diagonal being filled, starting along the diagonal and then moving up.

3 LCS

[CLRS 351-355] Given two sequences X and Y, we say that that Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y. The *longest common subsequence problem* asks you to find the maximum length common subsequence of two sequences X and Y.

Problem Description For two sequences $X =< x_1, x_2, \ldots, x_n >$ and $Y =< x_1, x_2, \ldots, x_n >$, $LCS(i,j)$ is the longest common subsequence of the sequences $X_i =< x_1, x_2, \ldots, x_i >$ and $Y_i = x_1, x_2, \ldots, x_j$.

Base Case $LCS(0,j) = LCS(i,0) = 0$

Recursive Solution (including base case)

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

Table Description LCS is an $n \times n$ table, the row index referring to the ending point of the first sequence X and the column index referring to the ending point of the second sequence Y. It is filled in row by row, from left to right(this is referred to as *row major order*). The running time to fill a table is only $O(nm)$ where n is the size of X and $m$is the size of Y.

4 Arithmetic Expression Parenthesization

You are given a sequence of n positive integers $x_1, x_2, \ldots, x_n$, alternating with $n-1$ arithmetic operators (either $+$ or $\times$), $op_1, op_2, \ldots, op_{n-1}$. This sequence corresponds to an arithmetic expression $RESULT = x_1\ op_1\ x_2\ op_2\ \ldots\ op_(n-1)\ x_n$. The problem is to fully *parenthesize* the arithmetic expression so that the $RESULT$ is *maximized*.

Problem Description $MAXP(i,j)$ is the maximum $RESULT_{i,j}$ achievable by fully parenthesizing the partial expression $x_i\ op_i\ x_{i+1}\ op_{i+1}\ \ldots\ op_(j-1)\ x_j$.

Base Case $MAXP(i,i) = X_i$

Recursive Solution (with base case)

$$MAXP(i,j) = \begin{cases} X_i & \text{if } i = j \\ \max_{i \le k < j}(MAXP(i,k)\ op_k MAXP(k+1,j)) & \text{otherwise} \end{cases}$$

Table Description MAP is an $n \times n$ table where the row index refers to the starting point of the subexpression and the column index refers to the ending point. Because of the max operation it takes $O(n^3)$ time to fill the array.

You need to drive along a highway, using rental cars. There are n rental car agencies 1, 2, ..., n along the highway. At any of the agencies, you can rent a car that can be returned at any other agency down the road. You cannot backtrack, i.e., you can drive only in one direction along the highway. So, a car rented at agency i can be returned only at some agency j ¿ i. For each pair of agencies i, j with j ¿ i, the cost of the i-to-j car rental is known.

Problem Definition Let $C(i, j)$ be the minimum cost of driving from rental agency i to rental agency j.

Base Case $C(i, i) = 0$

Recursive Solution (with base case)

$$C(i,j) = \begin{cases} 0 & i = j \\ \min_{i<k\leq j}(C(i,k) + C(k,j)) & i < j \end{cases}$$

Table Description C is an $n \times n$ table which, because of the min operation, it takes $O(n^3)$ to fill. We fill in the table in row-major order.

6 0-1 Knapsack

[CLRS 382] A thief robbing a store finds n types of items; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What is the maximum value of loot that he can remove from the store.

Problem Definition $MAX - KNAP(k)$ as the maximum value of loot attainable for a capacity k pounds.

Base Case $MAX - KNAP(0) = 0$

Recursive Solution (with base case)

$$MAX - KNAP(k) = \begin{cases} 0 & k = 0 \\ \max_i \text{ such that } {}_{w_i \leq k}[MAX - KNAP(k - w_i) + v_i] & otherwise \end{cases}$$

Table Description Here a one-dimensional table is being filled, where the index is k, the capacity. Since the max is being taken over all items, it takes $O(n)$ time to fill each element, when starting from the bottom up.