Dynamic Programming

1. Origin of term dynamic programming

This term originally had nothing to do with computer programming. It was devised by Richard Bellman, an American applied mathematician, to describe work he began in the 1940's and which culminated in 1953 with the term "dynamic programming", dynamic to mean solving one problem after another, and programming in the sense of finding an optimal program in the sense of a military training schedule. (Bellman was working for RAND Corp. under contract to the Air Force.)The problems are usually optimization problems, i.e., finding the best solution under certain constraints. [Some of the NP-complete problems we talked about have associated optimization versions, i.e., find the shortest Hamiltonian circuit in a graph, or find the largest complete subgraph of a graph, but to make them NP-complete we turned these optimization problems into decision problems, i.e., is there a Hamiltonian circuit of length k or less, is there a complete subgraph of size k or more.] In dynamic programming, the overall optimal solution can be found by finding optimal solutions to smaller problems that increase in size until the full solution is found. But today such problems are often best solved by computer programs as they can involve many computations, so now "dynamic programming" usually also means computer programming.

This approach is used to solve problems not only in mathematics and computer science, but also in the management science field (production planning and inventory control), economics (optimum consumption and saving), transportation (optimum train control), computational biology (matching gene sequences), and many more.

2. Characteristics of problems suitable for dynamic programming

a. Optimal subproblems

There must be a way to order subproblems (although not necessarily a linear ordering) so that "smaller" subproblems appear earlier in the ordering. And there must be a formula that finds an optimal solution to a subproblem if the optimal solutions to smaller subproblems are known.

b. Overlapping subproblems

There should be a (relatively small) number of distinct subproblems, although they may appear over and over (this is what "overlapping" means). The fact that they are used repeatedly means that their values can be computed once and stored so that they do not have to be recomputed over and over.

3. Fibonacci sequence

The famous Fibonacci sequence is defined as follows:

$F(1) = 1$
$F(2) = 1$
$F(n) = F(n - 1) + F(n - 2)$ for $n > 2$.

Suppose our task is to compute the nth Fibonacci value. This is not an optimization problem, but is simple enough to illustrate some ideas.

There is a clear ordering of Fibonacci values:

$$F(1), F(2), F(3) ,,,,, F(n-2), F(n-1), F(n)$$

and a given relationship to compute F(n) from the results of "smaller" subproblems:

$$F(n) = F(n-1) + F(n-2) \text{ for } n \geq 3$$

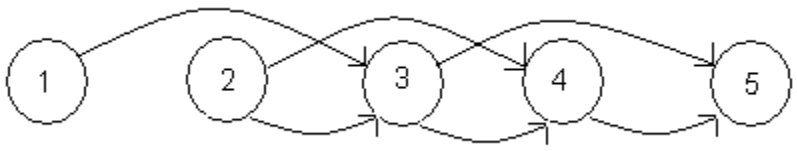
This satisfies characteristic (a) above.

As a "bottom-up" (iterative) approach, then, we compute F(1), F(2), F(3), etc., finally getting the value F(n) for a given n. Here's a description of this algorithm:

```
Fib(n)
{
      int counter = 3;         //counts up to n
      int one_back = 1;        //holds F(counter - 1)
      int two_back = 1;        //holds F(counter - 2)
      int next;                //next Fib value to compute

      if ((n == 1) || (n == 2))
            return 1;
      else
      {
            while (counter <= n)
            {
                  next = one_back + two_back;     //compute the next number
                  two_back = one_back;            //reset for the NEXT number
                  one_back = next;
                  counter = counter + 1;
            }
            return next;
      }
}
```

We can also think of this process as a directed graph. To compute F(5), for example, we walk left to right along the following directed graph, at each node after start-up using the values from the two previous nodes:



Or, we can imagine filling in entries in a 1-dimensional table, working from left to right. Each table entry for $i \geq 3$ is the sum of the two previous table entries.

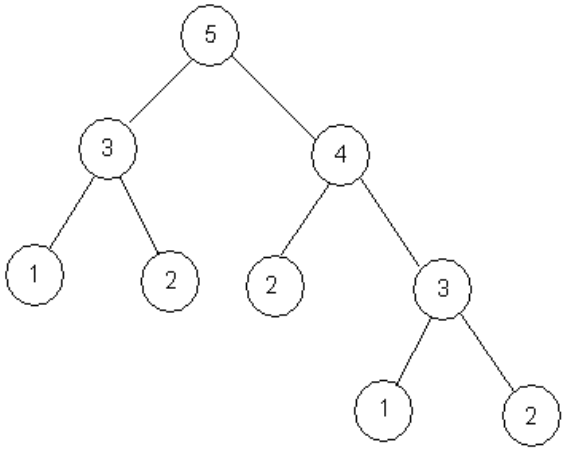
	i	1	2	3	4	5
F(i)		1	1	2	3	5

Looking at the code, which contains a simple while loop, the work done by this algorithm is clearly $\Theta(n)$.

Because of the recursive nature of the definition, it is natural to program a recursive solution to computing F(n):

```
recFib(n)
{
      if ((n == 1) || (n == 2))
            return 1;
      else
            return recFib(n - 1) + recFib(n - 2);
}
```

Consider what happens because of this "top-down", recursive approach to compute F(5).



The recFib function gets invoked 9 times, F(3) is computed twice, F(2) is computed 3 times, and F(1) twice. The general form of this diagram for an arbitrary n is a binary tree of height n – 2 where each interior node has two children. The leftmost branch of the tree is the shortest, with height ~n/2; cutting off the tree at this point would produce

$$1 + 2 + 2^2 + \ldots + 2^{n/2} = 2^{n/2 + 1} - 1$$

nodes (a lower bound). If the entire tree were completely full, there would be

$$1 + 2 + 2^2 + \ldots + 2^{n-2} = 2^{n-2 + 1} - 1 = 2^{n-1} - 1$$

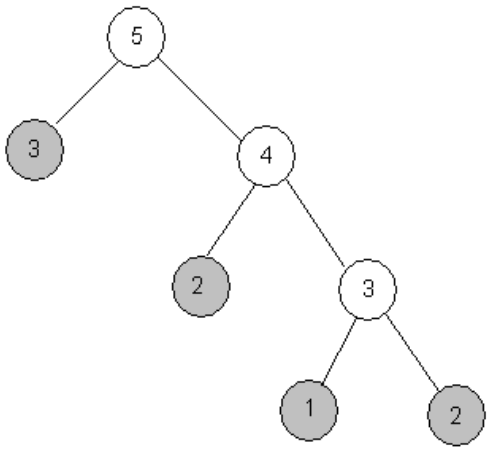
nodes (an upper bound). So we can see that the number of nodes is $\Theta(2^n)$, and that the recFib function gets invoked an exponential number of times. An exponential algorithm is never good! So the slick recursive function does a lot of work over and over and is not efficient. The Fibonacci problem clearly has subproblems that appear over and over, thus satisfying characteristic (b) above.

But – there is a way to salvage the top-down approach, which is to keep track of values already computed. Suppose that the first time F(k) is computed, its value gets stored in a table. Then the next time F(k) is needed, its value can just be looked up in the table instead of being recomputed. This technique is called *memoization* or *result caching*, and represents a classic time-space tradeoff. Some programming languages support memoization more or less automatically, while in other languages you have to modify your code to include the storage of new results and retrieval of previous results. Here's how you might use memoization where you have to manage it yourself.

```
int T[n]
for (i = 1..n)
        T[i] = 0;
T[1] = 1;
T[2] = 1;

memFib(n, T)
{
      if (!(T[n] == 0))
            return T[n];
      else
      {
            T[n] = memFib(n - 1, T) + memFib(n - 2, T);
            return T[n];
      }
}
```

The function memFib is still recursive, so let's see what happens when we compute F(5). The function calls look like this, where the shaded nodes are just table-lookups.



Comparing this with the previous tree, we see that the new algorithm, like the iterative algorithm, is again $\Theta(n)$, although it will have a higher constant coefficient than the original iterative algorithm because of the recursive calls and the table maintenance. In addition, we have a space requirement to store the table of previously-computed results.

One additional advantage accrues to the memoization method, however. Suppose we store the table of already computed values in persistent storage, i.e., in an external file. Then the next time we go to compute F(n) for a higher value (say F(8) instead of F(5)), we only have to compute F(6) and F(7) (and store them for future use); the rest is table look-up.

4. What about recursion?

Regular recursion (without memoization) seems to have gotten a bad name in our Fibonacci solution. Is recursion always bad? What about binary search, where in a sorted list the target is first compared against the middle element and, if not found, then either the front half or the back half is searched recursively? What about the mergesort sorting algorithm, where the list to be sorted is divided in half, each half is sorted recursively, and the two halves are merged together at the end?

In each of these two algorithms, each recursive call is invoked on a list only half the size of the previous list. Consequently in a list of size n, the maximum number of recursive calls is lg n. Progress is made in big bites. Contrast this with the Fibonacci recursion problem where each recursive call is made on a problem only slightly smaller than before, F(n – 1) or F(n – 2).

A divide-and-conquer approach, such as binary search or mergesort, is efficient because the division results in significantly smaller versions of the same problem. Consider the quicksort algorithm, which is also considered a divide-and-conquer approach. Quicksort works by selecting a pivot element from the list, throwing all the elements smaller than the pivot element in front of the pivot and all the larger elements behind it, then recursively sorting the "before" part and the "after" part. If the pivot element happens to be the median element (or close to it), then the two parts are relatively equal and close to size n/2, and the algorithm is efficient. If, however, the pivot element is the largest element (a possible worst case), then the "before" part is size n – 1 and the "after" part is empty, which changes the work of the algorithm from $\Theta(n \lg n)$ to $\Theta(n^2)$.

5. How to attack a dynamic programming problem

Now let's turn to the kinds of "optimization" problems that are best suited for dynamic programming (using a "bottom-up" approach of solving successively larger optimal subproblems). We've already mentioned that there should be an ordering to the smaller subproblems, a clear way to compute an optimal solution given optimal solutions to smaller subproblems, and that efficiency occurs if the values from subproblems can be stored and used repeatedly without recomputation.

We can often visualize the subproblems as entries in a table of some kind (possibly 2-dimensional) such that if we can figure out how to walk through the table in the "right" order, when we go to compute some entry of the table we will have all the "smaller" results available at that point. This worked for the Fibonacci example above.

The two examples given in the rest of this document follow a certain format.

What is the problem to be solved?

How would we solve it in a completely brute-force attack, and how expensive (how much work in the analysis of algorithms sense) would that be?

Can we identify smaller subproblems and a way to optimize the whole problem if we had optimized subproblem answers available?

Can we define an appropriate table structure to store already-computed values and an ordering for how to walk the table so that these values are available when needed?

Can we write a pseudocode algorithm to solve the problem?

What is the work involved?

However, it should be noted that the answers to these questions are often not obvious. That is, it may not be clear what the appropriate subproblems are or the order in which to solve them. So while dynamic programming is a very useful technique if you can figure out how to make it work for your problem, it's often not easy to do that and may require some trial and error.

6. Longest common subsequence

Given a sequence of characters $X = x_1, x_2, \ldots, x_m$, a **subsequence** of X is a sequence of (some) items from X in the same order in which they appear in X. They need not be contiguous items.

Example: If X = A, **B**, **C**, B, **D**, A, **B**, then Z = B, C , D, B is a subsequence of X consisting of the bold items from X.

Problem: Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, find the longest common subsequence of X and Y. One can imagine that such a problem is related to determining how closely two different gene sequences match.

Comments: Note that the problem is badly stated in that there may be several "longest" common subsequences. For example,

X = A, B, C, B, D, A, B Y = B, D, C, A, B, A

have several common subsequences of length 3: B, C, A and D, A, B and C, B, A. But none of these are a longest common subsequence because there are sequences of length 4: B, C, B, A and B, D, A, B. Length 4 is the longest common subsequence length. While a longest common subsequence need not be unique, the **length** of the longest subsequence is unique.

Brute-Force Solution: The brute-force approach is to take each subsequence from X and try to match it in Y. Given a subsequence $x_{i1}, x_{i2}, \ldots x_{ij}$ from X, we scan Y to see if this sequence also appears in Y. As a very rough cut at the number of comparisons required, we can imagine checking x_{i1} against all n characters in Y, then (assuming a success) checking x_{i2} against n – 1 characters in Y, etc. This would give something like $n + (n – 1) + \ldots 1$, which is $\Theta(n^2)$. This is an over-estimate, of course, because if we don't match x_{i1} until somewhere in the middle of the Y sequence, then we only have to check the remainder of the X subsequence against the remainder of the Y sequence, which is at this point shorter than n – 1. So perhaps we could reduce this $\Theta(n^2)$ expression, but it doesn't matter because that was the work for one particular subsequence

of X. How many subsequences are there? If X has length m, think of a binary string of length m where a 1 indicates that the X item at that position is part of the subsequence and a 0 indicates that the X item at that position is not part of the subsequence. Then the number of subsequences of X is the number of binary strings of length m, which is 2^m. So our work is now

some polynomial function of n $\times$ $\Theta(2^m)$

which is even worse than $\Theta(2^m)$. Again, an exponential algorithm (or worse) is never good!

Dynamic Programming: Does this problem lend itself to dynamic programming? It is an optimization problem – we want the longest common subsequence, that is, we want to maximize the common subsequence length property. A successful common subsequence of length k,

$Z = z_1, z_2, \ldots, z_k$

contains a common subsequence of length k – 1, $z_1, z_2, \ldots z_{k-1}$. Looking at this the other way around, from the common subsequence $z_1, z_2, \ldots z_{k-1}$, we have to find one more matching character to get $z_1, z_2, \ldots, z_k$ as a common subsequence. And we have to have at least one more X item and one more Y item available to be the last matching character. This seems to suggest an ordering of subproblems where we have to find common subsequences of prefixes of the X sequence and the Y sequence first.

Denote the prefix of X that is $x_1, x_2, \ldots x_i$ by X_i and use a similar notation on the Y sequence. X_0 (or Y_0) denotes an empty sequence. Then let L(i, j) be the length of the longest common subsequence of X_i and Y_j. This is a subproblem. We ultimately want L(m, n).

There are some trivial base cases. If either i = 0 or j = 0, then L(i, j) = 0 because you can't have a non-zero subsequence of an empty sequence. In general,

if $x_i = y_j$ then L(i, j) = L(i-1, j-1) + 1 //you can extend the subsequence to
// include the next common character

if $x_i \neq y_j$ then L(i, j) = max{L(i-1, j), L(i, j-1)} //you can't extend the subsequence, so
//go with the best you've got now

We have to solve three subproblems (L(i-1, j-1), L(i-1, j), L(i, j-1) to solve L(i, j). These subproblems each involve solving three smaller subproblems, etc. In short, in order to find L(m, n) we have to solve all the smaller L(i, j) problems. Because there are two variables, we can represent the results in a two-dimensional table. The base cases get us started by filling in row 0 and column 0 of the table. We can walk through the table in any order as long as the three smaller subproblems are solved first. In this case we can fill the table row by row.
The heart of the algorithm is therefore the following:

```
for (i = 0..m)
      L(i,0) = 0;
for (j = 0..n)
      L(0,j) = 0;
for (i = 1..m)
      for (j = 1..n)
            if (xi = yj)
                  L(i,j) = L(i-1,j-1) + 1;
            else if
                  L(i-1, j) >= L(i, j-1)
                        L(i,j) = L(i-1,j);
            else
                  L(i,j) = L(i,j-1);
```

In the table below, which illustrates the example case, the arrows show where succeeding table values **came from**. (The arrow in a given cell p pointing to a cell q means that the value for cell q came from cell p.) For example, L(4, 2) = 1 because x_4 and y_2 are not equal, so L(4, 2) is the maximum of either the previous row value or the previous column value. Those values are both 1, and from the way the pseudocode above is written, the value from the previous row is used in case of a tie. That's why there are so many more down arrows than right-pointing arrows in the table.

	j	0	1	2	3	4	5	6	
i		y_j	B	D	C	A	B	A	
0	x_i	0	0 ↓	0 ↓	0 ↓↘	0	0↘	0	
1	A	0 ↘	0	0	0	1→↓↘	1	1	
2	B	0	1→↓	1→↓↘	1	1	2→↓	2 ↓	
3	C	0 ↘	1	1↓	2→↓	2↓↘	2	2	
4	B	0	1↓↘	1	2↓	2 ↓	3→↓	3↓	
5	D	0	1↓	2↓	2↓↘	2	3↓↘	3	
6	A	0 ↘	1	2↓	2↓	3↓↘	3	4↓	
7	B	0	1	2	2	3	4	4	

Table 1

Finally, L(7, 6) = 4, which is the maximum length subsequence. Now the next question is, how do we actually find such a subsequence?

In the table, we start with L(7, 6) and walk the path back to the top or front edge. Any cell pointed to by a diagonal arrow represents a common subsequence item, i.e., a place where the x and y characters were equal.

	j	0	1	2	3	4	5	6	
i		y_j	B	D	C	A	B	A	
0	x_i	0	0 ↓	0 ↓	0 ↓↘	0	0↘	0	
1	A	0 ↘	0	0	0	1→↓↘	1	1	
2	B	0	1→↓	1→↓↘	1	1	2→↓	2 ↓	
3	C	0 ↘	1	1↓	2→↓	2↓↘	2	2	
4	B	0	1↓↘	1	2↓	2 ↓	3→↓	3↓	
5	D	0	1↓	2↓	2↓↘	2	3↓↘	3	
6	A	0 ↘	1	2↓	2↓	3↓↘	3	4↓	
7	B	0	1	2	2	3	4	4	

Table 2

The common subsequence for this path is therefore B, C, B, A. (Other candidates would be represented by alternate paths where the choice of which cell to use for the maximum in a tie value differs from what we used here.)

The code for the dynamic programming algorithm has been reduced to two nested for loops, $\Theta(mn)$, a big improvement over the brute-force exponential algorithm.

7. Matrix multiplication

In order to do matrix multiplication $A \times B$ (which I will write as AB), the number of columns of A must agree with the number of rows of B. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, then the product AB exists and is an $m \times p$ matrix. Furthermore, by the standard definition of matrix multiplication, this product requires mnp scalar multiplications (n multiplications of a row of A times a column of B for each of the mp positions in the resulting matrix). There is a slightly smaller number of additions required, so we can say that the work involved is $\Theta(mnp)$. Matrix multiplication is not **commutative**, that is, the product AB is not in general equal to the product BA. However, matrix multiplication is **associative**, that is, given matrices of appropriate dimensions,

$$(AB)(CD) = ((AB)C)D = A(B(CD))$$

etc. While the order of the matrices must stay the same, the order in which the multiplications are performed, determined by the grouping symbols, can vary.

Example: Matrices A, B, C, D have the following dimensions:

A: 30×1
B: 1×40
C: 40×10
D: 10×25

Several matrix multiplication orderings are shown below, together with the work they require.

1. ((AB)C)D $30 \times 1 \times 40 + 30 \times 40 \times 10 + 30 \times 10 \times 25 = 20{,}700$
2. A(B(CD)) $40 \times 10 \times 25 + 1 \times 40 \times 25 + 30 \times 1 \times 25 = 11{,}750$
3. (AB)(CD) $30 \times 1 \times 40 + 40 \times 10 \times 25 + 30 \times 40 \times 25 = 41{,}200$
4. A((BC)D) $1 \times 40 \times 10 + 1 \times 10 \times 25 + 30 \times 1 \times 25 = 1400$
5. (A(BC))D $1 \times 40 \times 10 + 30 \times 1 \times 10 + 30 \times 10 \times 25 = 8200$

So while the resulting matrix is the same in all five cases, the amount of work done varies wildly based on the parenthetical grouping (another example of the difference between "theory" and "practice.")

Problem: Given n matrices $A_1, A_2, \ldots A_n$, where the dimensions of A_i are $m_{i-1} \times m_i$, then the product of all n matrices can be computed. In the Example problem, this becomes

A	×	B	×	C	×	D
30×1		1×40		40×10		10×25
m_0 m_1		m_1 m_2		m_2 m_3		m_3 m_4

Find the parenthetical ordering that minimizes the cost of the matrix multiplication. This is clearly an important problem in complex engineering problems requiring many matrix multiplications.

Comment: As in the longest common subsequence problem, it is possible that the optimal ordering is not unique.

Brute-Force Solution: Examine all the possible parenthetical orderings, find the work for each, and pick the minimal one. But how many different parenthetical orderings are possible? We can (almost) solve this using a recurrence relation.

Let P(n) denote the number of ways to parenthesize the product of n matrices. There are two trivial base cases

P(1) = 1 there is only one way to "multiply" one matrix A, (A)
P(2) = 1 there is only one way to multiply AB, (AB)

For three or more matrices, the top-level parentheses represents the last multiplication to be done, and splits the list of matrices into 2 parts. The position at which such a split can occur ranges from 1 (after the first matrix) to n – 1 (before the nth matrix). From the example above, the top-level split occurs

1. between C and D, position 3
2. between A and B, position 1
3. between B and C, position 2
4. between A and B, position 1
5. between C and D, position 3

If the split occurs at position k, $1 \le k \le n-1$, there are then two sequences of matrices to be multiplied, one of size k and one of size n – k. These will have P(k) and P(n – k) parenthesizations, respectively, so for a fixed k, using the Multiplication Principle, there are P(k)P(n-k) parenthesizations possible. By the Addition Principle, we have to add the results for the various values of k. Therefore the general recurrence relation is

$P(1) = 1$

$P(2) = 1$

$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$ for $n \ge 3$

This recurrence relation looks very similar to the definition of the **Catalan sequence** of numbers:

$C(0) = 1$

$C(1) = 1$

$C(n) = \sum_{k=1}^{n} C(k-1)C(n-k)$ for $n \ge 2$

In fact, P(n) = C(n – 1) for all $n \ge 1$. We can prove this using the Second Principle of Induction:

$P(1) = 1 = C(0)$

$P(2) = 1 = C(1)$

Assume that for all r, $1 \le r \le m$, P(r) = C(r – 1). Then

$$P(m+1) = \sum_{k=1}^{m} P(k)P(m+1-k) = \sum_{k=1}^{m} C(k-1)C(m-k) = C(m)$$

Now it turns out that the C(n) recurrence relation has a closed-form solution [no proof here], where C(2n,n) = combinations of 2n things n at a time:

$$C(n) = \frac{1}{n+1}C(2n,n) = \frac{1}{n+1}\frac{(2n)!}{n!(2n-n)!} = \frac{1}{n+1}\frac{(2n)!}{n!n!} = \frac{(2n)!}{(n+1)!n!}$$

and that

$$\frac{(2n)!}{(n+1)!n!} \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

A limit test will prove that

$$\frac{4^n}{n^{3/2}\sqrt{\pi}}$$

is a larger order of magnitude than 2^n (of course we can ignore the constant square root of π) :

$$\lim_{n\to\infty}\frac{\left(\frac{4^n}{n^{3/2}}\right)}{2^n}=\lim_{n\to\infty}\frac{\left(\frac{(2^n)^2}{n^{3/2}}\right)}{2^n}=\lim_{n\to\infty}\frac{2^n}{n^{3/2}}=\infty$$

so the amount of work is bounded below by the exponential function 2^n, making it even worse than 2^n.

So trying all possible parenthetical orderings and computing the work for each is an unacceptable algorithm.

Dynamic Programming: Does this problem lend itself to dynamic programming? It is an optimization problem – we want to minimize the cost of the matrix multiplications. Can we find a sequence of subproblems whose optimal solution will ultimately lead us to the optimal solution to the overall problem?

Note that in a non-trivial case, that is, for $n \geq 3$, any particular parenthetical ordering p involves multiplication of smaller group(s) of matrices. If p is to be optimal, then each subgroup s must be optimally parenthesized. Otherwise, if there is a better way to parenthesize s, then using that would improve p.

Let M(i, j) represent the minimal work (scalar multiplications) required to multiply a group of matrices $A_iA_{i+1}\ldots A_j$ where $1 \leq i \leq j \leq n$. This is a subproblem. We ultimately want M(1, n).

As a trivial base case, if i = j, M(i, j) = M(i, i) = 0 (no work to "multiply" one matrix). To multiply $A_iA_{i+1}\ldots A_j$ where i < j, we again split this group at some point k, so that the multiplication is

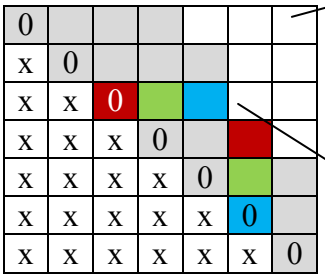
$$(A_iA_{i+1}\ldots A_k)(A_{k+1},\ldots Aj)$$

The minimum work M(i, j) for this product is M(i, k) + M(k+1, j) [remember these subproblems must be optimal] plus the work to multiply the two matrices resulting from these subproblems. These two matrices have dimensions of $m_{i-1} \times m_k$ and $m_k \times m_j$, so the work here is $(m_{i-1})(m_k)(m_j)$. So the work for this particular k is M(i, k) + M(k+1, j) + $(m_{i-1})(m_k)(m_j)$, but we need to minimize that over all possible k's. Therefore

$$M(i, j) = \min_{i\leq k<j}[M(i, k) + M(k+1, j) + (m_{i-1})(m_k)(m_j)] \quad \text{for } 1 \leq i < j \leq n$$

(k must be < j so that M(k+1, j) makes sense). At first glance this seems little better than the recurrence relation we had in the brute force approach. But much as we did for the longest subsequence problem, we can use a two-dimensional table (n × n) and walk our way through it from smaller to larger subproblems. The base case gets us started by filling in the main diagonal of the table with 0's because M(i, i) = 0. In addition, because i < j, we do not use the lower triangular part of the table (marked in the table below with x's). To compute M(i, j) we need to know the values of M(i, k) and M(k+1, j) for $i \leq k < j$. These are the table entries in row i to the left of column j (the current column), and in column j below row i (the current row). If we walk

through the table by diagonals from the main diagonal to the upper right, then we have the information we need for each entry, as shown in the table below. And we will end up at position M(1, n), the value we want.



Goal: M(1,n)

This is row i, column j. When we get here, all entries in this row to the left and in this column below are known and in the last line of the algorithm below they are paired up by the colors shown. (n = 7, d = 3, i = 3, j = 6, k = 3, 4, 5)

Table 3

In the algorithm below, d represents the successive diagonals, of which there are n – 1.

```
for (i = 1..n)
      M(i,i) = 0;
for (d = 1..n-1)
      for (i = 1..n-d)
            j = i + d;
            M(i,j) = min[M(i,k) + M(k+1,j) + (m_{i-1})(m_k)(m_j) with i ≤ k < j]
```

Alternatively, we can walk through the table from the bottom row (actually next to the bottom row) to the top row, and within each row from left to right.

You can see from Table 3 that there are less than n^2 entries in the table to fill. For each entry the min is computed in the last line of the algorithm, where k runs between i and j, so k is never more than n. Thus this algorithm is bounded above by n^3, that is, it is $O(n^3)$, which is a huge improvement over the brute-force approach.

Actually, there is an even better algorithm, O(n log n), published in 1984.[1] This paper does not solve the matrix multiplication problem directly, but converts it into a problem of partitioning a convex polygon into disjoint triangles, then finding an optimum solution for this partitioning problem. If you do a web search, you can find a 128-page version of this paper.

[1] Hu, T. C.. and Shing, M. T., *SIAM Journal on Computing,* 13 (2): 228–251.