

Notes on Dynamic Programming Algorithms & Data Structures 2007

Dr Mary Cryan

These notes are to accompany lecture 9 of ADS (2007).

1 Introduction

The technique of *Dynamic Programming (DP)* could be described “recursion turned upside-down”. However, it is not usually used as an *alternative* to recursion. Rather, dynamic programming is used (if possible) for cases when a recurrence for an algorithmic problem **will not** run in polynomial-time if it is implemented recursively. So in fact Dynamic Programming is a more-powerful technique than basic Divide-and-Conquer.

Designing, Analysing and Implementing a dynamic programming algorithm is (like Divide-and-Conquer) highly problem specific. However, there are particular features shared by most dynamic programming algorithms, and we describe them below. It will be helpful to carry along an introductory example-problem to illustrate these features. My introductory problem will be the problem of computing the n th Fibonacci number, where $F(n)$ is defined as follows:

$$\begin{aligned}F_0 &= 0, \\F_1 &= 1, \\F_n &= F_{n-1} + F_{n-2} \quad (\text{for } n \geq 2).\end{aligned}$$

Since Fibonacci numbers are defined recursively, the definition suggests a very natural recursive algorithm to compute $F(n)$:

Algorithm REC-FIB(n)

1. **if** $n = 0$ **then**
2. **return** 0
3. **else if** $n = 1$ **then**
4. **return** 1
5. **else**
6. **return** REC-FIB($n - 1$) + REC-FIB($n - 2$)

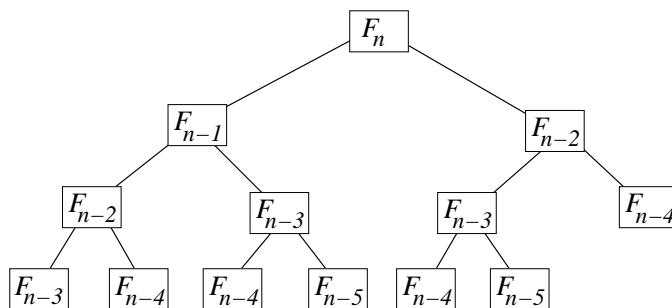
First note that were we to implement REC-FIB, we would not be able to use the Master Theorem to analyse its running-time. The recurrence for the running-time $T_{\text{REC-FIB}}(n)$ that we would get would be

$$T_{\text{REC-FIB}}(n) = T_{\text{REC-FIB}}(n - 1) + T_{\text{REC-FIB}}(n - 2) + \Theta(1), \quad (1)$$

where the $\Theta(1)$ comes from the fact that, at most, we have to do enough work to add two values together (on line 6). The Master Theorem cannot be used to analyse this recurrence because the recursive calls have sizes $n - 1$ and $n - 2$, rather than size n/b for some $b > 1$. The actual presence of non- n/b calls need not *necessarily* exclude the possibility of proving a good running-time

- for example, if we think back to the QUICKSORT algorithm (lecture 6), the worst-case recurrence for QUICKSORT contains a call of size $n - 1$. However, the *particular form* of (1) does in fact imply that $T_{\text{REC-FIB}}(n)$ is *exponential* in n , and therefore highly inefficient. It is easy to observe, looking at (1), that due to the similarity of the definition of $F(n)$ and the $T_{\text{REC-FIB}}(n)$ recurrence, we have $T_{\text{REC-FIB}}(n) \geq F(n)$. But unfortunately it is well-known that $F_n \approx 1.6^n$ as n gets large. We won't prove this here, but this asymptotic value for $F(n)$ is well-known.

A nice *illustration* of how running time of REC-FIB blows up exponentially can be seen by drawing the tree of Recursive calls made by a single call to REC-FIB. Notice in the diagram below that even after just 3 layers of the recursion tree, we see 4 calls to $n - 4$ (and there will be another call on the next level).



If you continue working this tree down a couple of levels, you'll see the number of calls increase as we get further from n .

Although the recursive algorithm has appalling performance, there is some hope. Although the recursion tree grows exponentially in size, it actually only contains $n + 1$ different instances of the problem: $F(n), F(n - 1), \dots, F(1), F(0)$. This is what will allow us to develop an efficient dynamic programming algorithm to evaluate $F(n)$ for a given n ¹. We now present a $\Theta(n)$ time dynamic programming algorithm for computing $F(n)$, with reference to the following standard features of a Dynamic programming algorithm:

dp1(a) The initial stage of designing a dynamic programming algorithm for a problem usually involves *generalising* the problem slightly, and solving a collection of smaller subproblems *as well as* the specific problem instance described by the input. Although this seems as though we have made our problem harder (by deciding to compute extra results), an intelligent choice of generalisation will be the key to solving our original problem efficiently.

For the *Fibonacci number* problem, our generalisation will be to compute all of $F(0), F(1), F(2), \dots, F(n)$, rather than simply $F(n)$.

dp1(b) Once we have generalised our problem, we will then write down a recurrence (or, more generally, a short algorithm) for solving one problem instance in terms of “smaller” (in some sense) problem instances. It must be the case that the smaller problem instances on the right-hand side of

¹Another way an efficient algorithm could be derived for Fibonacci numbers would be for the compiler to perform “memoization” and store pre-computed values for future use. However, while this is easy to detect in the context of the Fibonacci recurrence, it is not so obvious for more complex versions of DP.

the recurrence will lie within the set of subproblems identified in $dp1(a)$ above (and that this is true recursively). Hence $dp1(a)$ and $dp1(b)$ really must be done together.

For the Fibonacci number problem, our recurrence is as in the definition: $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$.

dp2 Next step is to allocate storage for the results for each of the subproblems which we will solve during our algorithm. These are the subproblems identified in $dp1(a)$, $dp1(b)$. Usually it is possible to store the subproblems in an table - hopefully, this will be 1-dimensional, or 2-dimensional, but there are problems which require 3-dimensional (or greater) tables. If we start designing a DP algorithm and find that the set of identified subproblems from $dp1$ will not fit in a sensibly-structured table with a reasonable number of dimensions, and of a reasonable size, this may in fact indicate that there is no good DP algorithm for that problem.

The table we use for the Fibonacci number problem is an array of length $n+1$, indexed from 0 upwards. This array will store $F(k)$ at the index k .

dp3 Finally, we must set up an algorithm to control the *order* in which subproblems are solved (and their results stored in the appropriate cell of the table). It is imperative that this be done in such a way that all of the subproblems appearing on the right-hand side of the recurrence must be computed and stored in the table *in advance* of the call to that recurrence (in cases where we can't see how to do this, it may be that our problem, or recurrence, is not amenable to DP).

Note that for Fibonacci numbers, the order is simple. We first fill in the array at index 0 and 1 (with values 0 and 1 respectively), and thereafter we compute and store $F(i)$ using the recurrence in the order $i = 2, \dots, n$.

We note that for the problem of computing Fibonacci numbers, we could in fact dispose of the table altogether, and just work with two variables `fib` and `fibm1`, representing the current Fibonacci number and the previous one. However, in more interesting examples of dynamic programming we always need a table (often of larger size/dimensions).

2 Matrix-chain Multiplication Problem

In the general setting of matrix multiplication, we may be given an entire list of rectangular matrices A_1, \dots, A_n , and wish to evaluate the product of this sequence. We assume that the matrices are rectangular, ie, each matrix is of the form $p \times q$ for some positive integers p, q . If the product of A_1, \dots, A_n is to exist at all, it must be the case that for each $1 \leq i < n$, the number of columns of A_i is equal to the number of rows of A_{i+1} . Therefore, we may assume that the dimensions of all n matrices can be described by a sequence $p_0, p_1, p_2, \dots, p_n$, such that matrix A_i has dimensions $p_{i-1} \times p_i$, for $1 \leq i \leq n$. Recall from Lecture 3 that Strassen's algorithm was designed for square matrices, not rectangular ones.² We therefore work in the setting where we always use the naïve

²There is a simple adaption of Strassen's algorithm for rectangular matrices, but it is only an improvement on the naïve algorithm if p, q and r are very close in value. Therefore it is

“high-school” algorithm to multiply pairs of matrices. Given any two rectangular matrices A and B of dimensions $p \times q$ and $q \times r$ respectively, the naïve algorithm requires pqr number multiplications to evaluate the product AB .

We now give an example which shows that when we have an entire sequence of matrices to multiply together, the way we “bracket” the sequence can have a huge effect on the total number of multiplications we need to perform.

Example:

Suppose our set of input matrices have dimensions as follows:

$$\begin{array}{cccc} A & \cdot & B & \cdot & C & \cdot & D \\ 30 \times 1 & & 1 \times 40 & & 40 \times 10 & & 10 \times 25 \end{array}$$

If we were to multiply the sequence according to the order $(A \cdot B) \cdot (C \cdot D)$ we would use

$$30 \cdot 1 \cdot 40 + 40 \cdot 10 \cdot 25 + 30 \cdot 40 \cdot 25 = 41,200$$

multiplications. The initial term $30 \cdot 1 \cdot 40$ is the number of multiplications to get AB , which will then have dimensions 30×40 . The term $40 \cdot 10 \cdot 25$ is the number of mults. to evaluate CD , which has dimensions 40×25 . Then from these two results, we can get $A \cdot B \cdot C \cdot D$ using $30 \cdot 40 \cdot 25$ extra multiplications.

However, if we instead use the parenthesisation $A \cdot ((B \cdot C) \cdot D)$ to guide our computation, we use only 1,400 multiplications to compute $A \cdot B \cdot C \cdot D$:

$$1 \cdot 40 \cdot 10 + 1 \cdot 10 \cdot 25 + 30 \cdot 1 \cdot 25 = 1,400$$

This example motivates the problem of finding an optimal parenthesisation (“bracketing”) for evaluating the product of a sequence of matrices, called the *Matrix-chain Multiplication Problem*:

Input: Sequence of matrices A_1, \dots, A_n , where A_i is a $p_{i-1} \times p_i$ -matrix

Output: Optimal number of multiplications needed to compute $A_1 A_2 \dots A_n$, and an optimal parenthesisation

Note that we would expect the Matrix-chain Multiplication Problem to be solved well in advance of doing any matrix multiplication, as the solution will tell us how the matrix multiplications should be organised. Therefore we only really need to give p_0, p_1, \dots, p_n as the input. The running time of our algorithms will therefore be measured in terms of n .

2.1 Solution Attempts (which don’t work)

Before we attack the problem using Dynamic Programming (or even Divide-and-Conquer), we mention certain algorithms/strategies which are inefficient and/or non-optimal.

Approach 1: Exhaustive search.

This approach involves enumerating all possible parenthesisations, evaluating

not appropriate to consider this adapted version of Strassen in the general setting where we expect to have wildly varying row-counts and column-counts among the A_i matrices.

the number of multiplications for each one. Then the best is taken. This approach is correct, but extremely slow - it can be shown that the running time is $\Omega(3^n)$, because there are $\Omega(3^n)$ different parenthesisations. I've put a proof of the $\Omega(3^n)$ result on the webpage - however, it is tricky; if I was asking you to prove something, I'd only ask for $\Omega(2^n)$.

Approach 2: Greedy algorithm.

This version of a greedy algorithm involves doing the cheapest multiplication first. This approach would run in $O(n \lg(n))$ time (by sorting of the $p_{i-1} \cdot p_i \cdot p_{i+1}$ values and intelligently updating the sorted list after each decision is made). However, this technique does *not* work correctly — sometimes, it returns a parenthesisation that is not optimal, as in the following example:

$$\begin{array}{ccccc} A_1 & \cdot & A_2 & \cdot & A_3 \\ 3 \times 100 & & 100 \times 2 & & 2 \times 2 \end{array}$$

The solution proposed by the greedy algorithm is $A_1 \cdot (A_2 \cdot A_3)$. Evaluating the product this way uses $100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000$ multiplications. However, the optimal parenthesisation is $(A_1 \cdot A_2) \cdot A_3$ which uses $3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612$ multiplications.

Two alternative approaches, neither of which is guaranteed to give the optimal parenthesisation (though both are reasonably fast), are:

- Set the outermost parentheses so that the cheapest multiplication gets done last (alternative greedy).
- Choose the initial pairing of two neighbouring matrices $A_i A_{i+1}$ to maximize the number of columns of A_i (this p_i will disappear immediately after the computation $A_i A_{i+1}$ is performed).

Try to find examples where the techniques above fail to return an optimal parenthesisation.

2.2 Dynamic Programming solution

We now consider how we might set up a recurrence to solve the Matrix-chain Multiplication problem (this is dp1(b) from our list of features of DP). A very simple observation is that every parenthesisation must have *some* top-level parenthesis - the multiplication which is “done last”. Therefore one way of partitioning the set of all parenthesisations is according to the matrix index where the top-level parenthesisation is done:

$$(A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n)$$

Note that in this setting, if we consider a top-level parenthesisation just after A_k , the number of mults. done for the final computation is $p_0 p_k p_n$. *More importantly*, note that deciding to focus on having the break after A_k imposes a natural “Divide-and-Conquer” structure on the problem - we can find the optimal parenthesisation of $A_1 \cdot A_2 \dots A_k$ independently of the optimal parenthesisation of $A_{k+1} \dots A_n$ (once we have decided the top-level break is after A_k).

This gives a natural recursive algorithm to compute the optimal parenthesisation of $A_1 \dots A_n$. We can consider all possible positions k for the top-level

break in turn and for each such k , recursively solve the two sub-problems. First we will define, for each $1 \leq i \leq j \leq n$,

$$m[i, j] = \text{least number of multiplications needed to compute } A_i \cdots A_j$$

Now we identify the generalised version of the Matrix-chain multiplication problem which we will solve (remember this is one of the initial steps in designing a DP algorithm):

dp1(a) Given a list of matrices A_0, A_1, \dots, A_n and their dimensions p_0, p_1, \dots, p_n , we will compute the value $m[i, j]$, for *every* $1 \leq i < j \leq n$ (naturally including $i = 1, j = n$, which is the original target problem).

dp1(b) We will now write down the recurrence which our dynamic programming algorithm will use to compute the $m[i, j]$ values. Our basic observation about considering a position k for the top-level break, and then using Divide-and-Conquer for the left and right sides that result, gives us the following recurrence:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{1 \leq k < j} (m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j) & \text{if } i < j. \end{cases} \quad (2)$$

Note that the recurrence in dp1(b) has motivated the choice of which collection of subproblems we choose to solve in dp1(a). An initial examination of the recurrence for $m[1, n]$ will only generate subproblems of the form $m[1, k]$ or of the form $k + 1, n]$, ie, subproblems which include A_1 or A_n - however, going down more levels, we see that we may need to look up $m[i, j]$ for arbitrary $1 \leq i < j \leq n$. Note that this set of subproblems is more interesting than the set for $F(n)$, in two ways:

- The subproblems are indexed 2-dimensionally rather than 1-dimensionally.
- The subproblems depend on the actual input values (the p_i, p_{i+1}, \dots, p_j), rather than just on the size of the input.

We now discuss the two remaining DP issues:

dp2 The basic dynamic programming table is very simple to describe. It will be a $n \times n$ table named m , with rows indexed by $1 \leq i \leq n$ and columns by $1 \leq j \leq n$. The cell $m[i, j]$ is intended to store the value of the optimal parenthesisation of $A_i \dots A_j$ (note we can assume $m[i, j] = \infty$, or that $m[i, j]$ is undefined, for all $i > j$).

Recall that our dynamic programming algorithm is required to compute a *structure* (some particular optimal parenthesisation) as well as the *value* of an optimal parenthesisation.³ We will define a second table s of size $n \times n$, where $s[i, j]$ will store the value k of the (minimum) “top-level break” ($i \leq k < j$) which will achieve a optimal parenthesisation for $A_i \dots A_j$. Formally,

$$s[i, j] = (\text{the smallest}) k \text{ such that } i \leq k < j \text{ and } m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j.$$

³Depending on the particular problem being solved, you may just need to compute a value, or also find a structure realising that value. In the latter case, an extra table is usually needed.

dp3 Next we need to prescribe the order in which the table will be filled-in. To do this examine the recurrence (2). Note that when we consider $i \leq k < j$, we are guaranteed that both the “left-side” and the “right-side” will contain at least one matrix. Hence for any k considered, the subproblems referenced contain *strictly* fewer matrices than the current problem under consideration. Therefore we only need organise our algorithm to compute the solutions for shorter sequences of matrices first, and once we have done that, we are guaranteed that all values on the right-hand side of the recurrence will be available when they are needed.

So as usual, the key to getting a good DP algorithm is to compute solutions in a bottom-up fashion.

Our algorithm will first initialise $m[i, i] = 0$ for all $i, 1 \leq i \leq n$ (considering all sequences of length 1). Next it will consider all sequences $A_i A_{i+1}$ of length 2, in order of i , and so on. This can be achieved by an outer loop controlling the length of sequences ($\ell = 1, 2, \dots, n$), followed by an inner loop controlling the starting point of sequences of the current length. Finally, for each ℓ and i , the algorithm will consider at most $\ell - 1 < n - 1$ different possible breakpoints. Hence the running time will be $O(n^3)$. The algorithm will also update the $s[i, j]$ pointers at the same time as it fills in the m table.

See MATRIX-CHAIN-ORDER below for the details.

Algorithm MATRIX-CHAIN-ORDER(p)

1. $n \leftarrow p.length - 1$
2. **for** $i \leftarrow 1$ **to** n **do**
3. $m[i, i] \leftarrow 0$
4. **for** $\ell \leftarrow 2$ **to** n **do**
5. **for** $i \leftarrow 1$ **to** $n - \ell + 1$ **do**
6. $j \leftarrow i + \ell - 1$
7. $m[i, j] \leftarrow \infty$
8. **for** $k \leftarrow i$ **to** $j - 1$ **do**
9. $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
10. **if** $q < m[i, j]$ **then**
11. $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. **return** s

Running Time: The running time of MATRIX-CHAIN-ORDER is $\Theta(n^3)$. The $O(n^3)$ is fairly straightforward. If you need help for $\Omega(n^3)$ ask me or your tutor.

Example

$$A_1 \quad \cdot \quad A_2 \quad \cdot \quad A_3 \quad \cdot \quad A_4$$

$$30 \times 1 \quad \quad 1 \times 40 \quad \quad 40 \times 10 \quad \quad 10 \times 25$$

Solution for m and s

m	1	2	3	4
1	0	1200	700	1400
2		0	400	650
3			0	10 000
4				0

s	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

Optimal Parenthesisation

$$A_1 \cdot ((A_2 \cdot A_3) \cdot A_4)$$

Multiplying the Matrices (using the s table)

Algorithm MATRIX-CHAIN-MULTIPLY(A, p)

1. $n \leftarrow A.length$
2. $s \leftarrow \text{MATRIX-CHAIN-ORDER}(p)$
3. **return** REC-MULT($A, s, 1, n$)

Algorithm REC-MULT(A, s, i, j)

1. **if** $i < j$ **then**
2. $C \leftarrow \text{REC-MULT}(A, s, i, s[i, j])$
3. $D \leftarrow \text{REC-MULT}(A, s, s[i, j] + 1, j)$
4. **return** $(C) \cdot (D)$
5. **else**
6. **return** A_i

2.3 Analysis of the Recursive Algorithm

For the Matrix-chain multiplication problem, I never went into depth about why the recurrence (2) cannot be used to design a straightforward Divide-and-Conquer algorithm. The reason is that, as with the $F(n)$ recurrence, the running-time of the natural Divide-and-Conquer algorithm is exponential in n . Note that for a recursive algorithm REC-MATRIX-CHAIN, the running time $T_{R-M-C}(n)$ satisfies the recurrence

$$T_{R-M-C}(n) = \sum_{k=1}^{n-1} (T_{R-M-C}(k) + T_{R-M-C}(n-k)) + \Theta(n).$$

This implies

$$T_{R-M-C}(n) = \Omega(2^n).$$

The easiest way to see $T_{R-M-C}(n) = \Omega(2^n)$ is to prove $T_{R-M-C}(n) \geq 2^{n-1}$ (assuming $T(1) = 1$)..

Mary Cryan