Chapter B2: Dynamic Programming

Often there is no way to divide a problem into a small number of subproblems whose solution can be combined to solve the original problem. In such cases we may attempt to divide the problem into many subproblems, then divide each subproblem into smaller subproblems and so on. If this is all we do, we will most likely end up with an exponential-time algorithm.

Often, however, there is actually a limited number of possible subproblems, and we end up solving a particular subproblem several times. If instead we keep track of the solution to each subproblem that is solved, and simply look up the answer when needed, we would obtain a much faster algorithm.

In practice it is often simplest to create a table of the solutions to all possible subproblems that may arise. We fill the table not paying too much attention to whether or not a particular subproblem is actually needed in the overall solution. Rather, we fill the table in a particular order.

B2.1 Longest Increasing Subsequence

Consider the problem of: Given a sequence of \( n \) values, find the longest increasing subsequence. By subsequence we mean that the values must occur in the order of the sequence, but they need not be consecutive.

For example, consider 3,4,1,8,6,5. In this case, 4,8,5 is a subsequence, but 1,3,4 is not. The subsequence 4,8 is increasing, the subsequence 3,4,1 is not. Out of all the subsequences, we want to find the longest one that is increasing. In this case this is 3,4,5 or 3,4,8 or 3,4,6.

Now we want an efficient algorithm for this. One idea is to try all subsequences. But there are too many. Divide-and-conquer does not appear to work either.

Instead we need an idea. And the idea is this: if we know the longest increasing subsequence that ends at 3, and the longest one that ends at 4, and \( \ldots \) the longest one that ends at 6, then we can determine the longest that ends at 5.

How? Well, any increasing subsequence that ends at 5 has a penultimate element that is smaller than 5. And to get the longest increasing subsequence that ends at 5 with, for example, penultimate 4, one takes the longest increasing subsequence that ends with 4 and appends 5.

Books on dynamic programming talk about a principle of optimal substructure: paraphrased, this is that

- “a portion of the optimal solution is itself an optimal solution to a portion of the problem.”
For example, in our case we are interested in the longest increasing sequence; call it \( L \). If the last element of \( L \) is \( A[m] \), then the rest of \( L \) is the longest increasing subsequence of the sequence consisting only of those elements lying before \( A[m] \) and smaller than it.

Suppose the input is array \( A \). Let \( f(m) \) denote the longest increasing subsequence that ends at \( A[m] \). In our example:

\[
\begin{array}{c|cccccc}
A & 3 & 4 & 1 & 8 & 6 & 5 \\
f & 1 & 2 & 1 & 3 & 3 & 3 \\
\end{array}
\]

In general, to compute \( f(m) \): go through all the \( i < m \), look at each \( i \) such that \( A[i] < A[m] \), determine the maximum \( f(i) \), and then add 1. In other words:

\[
f(m) = 1 + \max_{i<m}\{ f(i) : A[i] < A[m] \}
\]

Efficiency. The calculation of a particular \( f(m) \) takes \( O(m) \) steps. The algorithm calculates \( f(1) \), then \( f(2) \), then \( f(3) \) etc. Thus, the total work is quadratic—it’s on the order of \( 1 + 2 + \cdots + n \).

Dynamic programming has a similar flavor to divide-and-conquer. But dynamic programming is a bottom-up approach: smaller problems are solved and then combined to solve the original problem. The efficiency comes from storing the intermediate results so that they do not have to be recomputed each time.

### B2.2 Largest Common Subsequence

The largest increasing subsequence problem discussed above is a special case of the **largest common subsequence problem**. In this problem, one is given two strings or arrays and must find the longest subsequence that appears in both of them.

(Explain why the longest increasing subsequence problem is a special case of the longest common subsequence problem.)

The approach is similar to above. One does organized iteration. Suppose the input is two arrays \( A \) and \( B \). Then define

\[ g(m, n) \text{ to be the longest common subsequence that ends with } A[m] \text{ and } B[n]. \]

Obviously this is 0 unless \( A[m] = B[n] \).

To compute \( g(m, n) \) from previous information, we again look at the penultimate value in the optimal subsequence. Say the penultimate is in position \( i \) in \( A \) and in position \( j \) in \( B \). Then:

\[
g(i, j) = \begin{cases} 
0 & \text{if } A[i] \neq B[j] \\
\max \{ g(i-1, j), g(i, j-1) \} + 1 & \text{if } A[i] = B[j] 
\end{cases}
\]

Thus, the total work is quadratic—it’s on the order of \( 1 + 2 + \cdots + n \).
in $B$ (with of course $A[i] = B[j]$). Then the portion up to the penultimate is the longest common subsequence that ends with $A[i]$ and $B[j]$.

So we obtain the recursive formula:

$$g(m, n) = \begin{cases} 
1 + \max_{i<m, j<n} g(i, j) & \text{if } A[m] = B[n], \\
0 & \text{otherwise} 
\end{cases}$$

A simple implementation yields an $O(n^3)$ algorithm. Note that rather than doing recursion, one works one’s way systematically through the table of $g(m, n)$ starting with $g(1, 1)$, then $g(2, 1)$, then $g(3, 1)$, and so on.

### B2.3 The Triangulation Problem

The following is based on the presentation of Cormen, Leiserson, and Rivest.

A **polygon** is a closed figure drawn in the plane which consists of a series of line segments, where two consecutive line segments join at a vertex. It is called a **convex polygon** if no line segment joining a pair of nonconsecutive vertices intersects the polygon.

To form a **triangulation** of the polygon, one adds line segments inside the polygon such that each interior region is a triangle. If the polygon has $n$ vertices, $n - 3$ line segments will be added and there will be $n - 2$ triangles. (Why?)

![Triangulation diagram](image)

Now, suppose that associated with any possible triangle there is a **weight** function. For example, one might care about the perimeter of the triangle. Then the **weight** of a triangulation is the sum of the weights of the triangles. An **optimal triangulation** is one of minimum weight.

(This problem comes up in graphics. In particular it is useful to find the optimal triangulation where the weight function is the perimeter of the triangle.)

There is a recursive nature to the optimal triangulation. But one still has to be careful to ensure that the number of subproblems does not become exponential.

Assume that the vertices of the polygon are labeled $v_1, v_2, \ldots, v_n$ where $v_n$ is adjacent to $v_1$. Now, the side $v_1v_n$ must be the side of some triangle: say $v_k$ is the
other vertex of the triangle in the optimal triangulation. This triangle splits the polygon into two smaller polygons: one with vertices \( \{v_1, v_2, \ldots, v_k\} \) and one with vertices \( \{v_k, v_{k+1}, \ldots, v_n\} \). Furthermore, the optimal triangulation of the original problem includes optimal triangulations of these two smaller polygons.

![Diagram of triangle triangulation](image)

Now, if we apply the recursion to the smaller polygons using the non-original segment as the base of the triangle, we get two smaller problems, and the boundaries of these polygons again only have one non-original segment.

Let us define \( t[i, j] \) for \( 1 \leq i < j \leq n \) as the weight of an optimal triangulation of the polygon with vertices \( \{v_i, v_{i+1}, \ldots, v_j\} \). If \( i = j - 1 \) then the polygon is degenerate (has only two vertices) and the optimal weight is defined to be 0.

When \( i < j - 1 \), we have a polygon with at least three vertices. We need to minimize over all vertices \( v_k \), for \( k \in \{i+1, i+2, \ldots, j-1\} \), the weight of the triangle \( v_i v_k v_j \) added to the weights of the optimal triangulations of the two smaller polygons with vertices \( \{v_i, v_{i+1}, \ldots, v_k\} \) and \( \{v_k, v_{k+1}, \ldots, v_j\} \).

Thus we obtain the formula:

\[
t[i, j] = \begin{cases} 
0 & \text{if } i = j - 1 \\
\min_{i < k < j} t[i, k] + t[k, j] + w(\triangle v_i v_k v_j) & \text{if } i < j - 1
\end{cases}
\]

The calculation of the time needed is left to the reader.

**B2.4 Matrix Chain Multiplication**

Recall that the product \( AB \) of two matrices \( A \) and \( B \) is valid exactly when the number of columns of \( A \) equals the number of rows of \( B \). The result has the number of rows of \( A \) and the number of columns of \( B \).

To multiply a sequence of matrices, such as \( ABCD \), the number of columns of one matrix must equal the number of rows of the next matrix. If that condition is valid, then one can parenthesize the sequence into two-matrix products. For example, to
compute $ABCD$, we could proceed $((AB)C)D$ or $(AB)(CD)$ for example. It is known that matrix multiplication is associative: the choice of parentheses does not affect the answer. (It is not commutative though, so reordering is not allowed.)

However, some of these choices are more efficient than others. We consider here the problem of:

what choice of parentheses gives the fewest scalar multiplications?

We assume one uses the naive method for matrix multiplication (not, for example, Strassen). Recall that naive multiplication of an $m \times n$ matrix by an $n \times \ell$ matrix takes exactly $mn\ell$ scalar multiplications.

So the input is just the dimensions: say $p_0, p_1, \ldots, p_n$ where matrix $A_i$ has dimensions $p_{i-1} \times p_i$. The goal is the optimal parenthesization to compute the product $A_1A_2\ldots A_n$.

It turns out that there are (very crudely) about $4^n$ possible parenthesizations. But this problem can be solved efficiently by dynamic programming. The key is that the sub-parenthesizations must themselves be optimal. (Huh?)

Let me explain. At the very last multiplication, we will multiply two matrices. This will be $(A_1\ldots A_k)(A_{k+1}\ldots A_n)$ for some $k$. We don’t know yet what $k$ is. But one can observe that $A_1\ldots A_k$ should be calculated as cheaply as possible, as should $A_{k+1}\ldots A_n$, and that how they are calculated does not affect how many scalar multiplications are needed when they are finally multiplied. Now think recursion. We don’t know $k$: so try all possible $k$. To determine the optimal parenthesization for $A_1\ldots A_k$, again the final step is a product of two matrices, say $(A_1\ldots A_j)(A_{j+1}\ldots A_k)$. Looking some more, one can see that every problem in the recursion has the same structure...

So here is the algorithm: define $m[i,j]$ to be the minimum number of scalar multiplications to form the product $A_i\ldots A_j$. The product $A_i\ldots A_j$ will be computed by computing $A_i\ldots A_k$ and $A_{k+1}\ldots A_j$ and then multiplying the two results together. This means that the recurrence is (wait for it)

$$m[i, j] = \min \{ m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j \}$$

where the minimum is over all $k$ between $i$ and $j$. We need to be a bit more precise on the range of $k$. If we define $m[i, i]$ to be 0, then we are okay if we say that $k$ runs from $i$ up to $j - 1$ (inclusive).

The code ends up filling a table of $m[i,j]$ with $i \leq j$. In particular, we fill the table diagonal by diagonal. First we do $m[i, j]$ where $j - i = 1$, then where $j - i = 2$, and so on. The final value is in $m[1,n]$.

Here is an example: suppose the $p_i$ are 6, 5, 3, 1, 2, 4, 2. Then the resulting table is as
follows:

\[
\begin{pmatrix}
0 & 90 & 45 & 57 & 77 & 73 \\
0 & 15 & 25 & 43 & 41 \\
0 & 6 & 20 & 22 \\
0 & 8 & 16 \\
0 & 16 \\
0 &
\end{pmatrix}
\]

For example, \(m[2, 5] = 43\) is calculated as the minimum of
\[m[2, 2] + m[3, 5] + 5 \times 3 \times 4 = 0 + 20 + 60 = 80\]
\[m[2, 3] + m[4, 5] + 5 \times 1 \times 4 = 15 + 8 + 20 = 43\]
\[m[2, 4] + m[5, 5] + 5 \times 2 \times 4 = 25 + 0 + 40 = 65.\]

The optimal parenthesization takes \(m[1, 6] = 73\) scalar multiplications. It can be determined to be \((A_1(A_2A_3))(A_4A_5)A_6)\) (and is unique).

**Exercises**

1. Illustrate the behavior of:
   a) The longest increasing subsequence algorithm on the list:
   2, 11, 3, 10, 8, 6, 7, 9, 1, 4, 5
   b) The longest common subsequence algorithm on the two lists:
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 and 2, 11, 3, 10, 8, 6, 7, 9, 1, 4, 5

2. Code up the longest common subsequence algorithm. Your algorithm should return one of the longest common subsequences, not just the length.

3. John Doe wants an algorithm to find a triangulation where the sum of the lengths of the additional line segments is as small as possible. Can you help? Jane Doe wants an algorithm to find an optimal triangulation where the weight function is the area of the triangle. Can you help?

4. (*From Cormen et al.*) Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of \(n\) words of lengths \(l_1, l_2, \ldots, l_n\), measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of \(M\) characters each. Our criterion of “neatness” is as follows. If a given line contains words \(i\) through \(j\) and we leave exactly one space between words, the number of extra characters at the end of the line is \(M - j + i - \sum_{k=i}^{j} l_k\). The **penalty** for that line is the cube of the number of extra spaces. We wish to minimize the sum, over all lines except the last, of the penalties. Give a dynamic-programming algorithm to print a paragraph of \(n\) words neatly on a printer. Analyze the running time and storage requirements of your algorithm.