1. One-dimensional dynamic programming

String Reconstruction. Suppose that all blanks and punctuation marks have been inadvertently removed from a text file, and its beginning was polluted with a few extraneous characters, so the file looks something like "lioneuponatimeinafarfarawayland..." You want to reconstruct the file using a dictionary.

This is a typical problem solved by dynamic programming. We must define what is an appropriate notion of subproblem. Subproblems must be ordered by size, and each subproblem must be easily solvable, once we have the solutions to all smaller subproblems. Once we have the right notion of a subproblem, we write the appropriate recursive equation expressing how a subproblem is solved based on solutions to smaller subproblems, and the program is then trivial to write. The complexity of the dynamic programming algorithm is precisely the total number of subproblems times the number of smaller subproblems we must examine in order to solve a subproblem.

In this and the next few examples, we do dynamic programming on a one-dimensional object—in this case a string, next a sequence of matrices, then a set of strings alphabetically ordered, etc. The basic observation is this: A one-dimensional object of length \( n \) has about \( n^2 \) sub-objects (substrings, etc.), where a sub-object is defined to span the range from \( i \) to \( j \), where \( i, j \leq n \). In the present case a subproblem is to tell whether the substring of the file from character \( i \) to \( j \) is the concatenation of words from the dictionary. Concretely, let the file be \( f[1...n] \), and consider a 2-D array of Boolean variables \( T(i, j) \), where \( T(i, j) \) is true if and only if the string \( f[i...j] \) is the concatenation of words from the dictionary. The recursive equation is this:

\[
T(i, j) = \text{dict}(x[i...j]) \lor \bigvee_{i \leq k < j} T(i, k) \land T(k + 1, j)
\]

In principle, we could write this equation verbatim as a recursive function and execute it. The problem is that there would be exponentially many recursive calls for each short string. dynamic programming can be seen as a technique of implementing such recursive programs with heavy overlap between the recursion trees of the two recursive calls, so that the recursive function is called once for each argument. This is done by modifying the recursive program so that, before each recursive call a table is consulted. If the answer is in the table, the recursive call is canceled. At the end of each recursive call, the table is updated. Unwinding the recursion, we get:

```plaintext
for \( d := 1 \) to \( n - 1 \) do  (d is the difference between \( i \) and \( j \),
     for \( i := 1 \) to \( n - d \) do  (the size of teh subproblem currently solved)
         if \( \text{dict}(x[i...j]) \) then \( T(i, j) := \text{true} \) else
             for \( k := i \) to \( i + d - 1 \) do
                 if \( T(i, k) = \text{true} \) and \( T(k + 1, j) = \text{true} \) then do \{ \( T(i, j) := \text{true} \) \}
The complexity of this program is \( O(n^3) \): Three nested loops, ranging each roughly over \( n \) values.

Naturally, this program just returns a meaningless Boolean, and does not tell us how to reconstruct the text. Expanding the innermost loop (the last assignment statement) to

\{ \( T[i, j] := \text{true} \), first[\( i, i + d \) := k], exit for \}

where first is an array of pointers initialized to \( \text{nil} \), gives us also the end of the first word of each substring that is indeed the concatenation of dictionary words. Notice that this improves the running time, by exiting the for loop after the first match; more optimizations are possible.
This is typical of dynamic programming algorithms: Once the basic algorithm has been derived using dynamic programming, clever modifications that exploit the structure of the problem speed up its running time.

**Chain matrix multiplication**

Suppose next that you want to multiply four matrices $A \times B \times C \times D$ of dimensions $40 \times 20$, $20 \times 300$, $300 \times 10$, and $10 \times 100$, respectively. Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $mnp$ multiplications (a good enough estimate of the running time).

To multiply these matrices as $(((A \times B) \times C) \times D)$ takes $40 \cdot 20 \cdot 300 + 40 \cdot 300 \cdot 10 + 40 \cdot 10 \cdot 100 = 380,000$. A more clever way would be to multiply them as $(A \times ((B \times C) \times D))$, with total cost $20 \cdot 300 \cdot 10 + 20 \cdot 10 \cdot 100 + 40 \cdot 20 \cdot 100 = 160,000$. An even better order would be $((A \times (B \times C)) \times D)$ with total cost $20 \cdot 300 \cdot 10 + 40 \cdot 20 \cdot 10 + 40 \cdot 10 \cdot 100 = 108,000$. Among the five possible orders (the five possible binary trees with four leaves) this latter method is the best.

How can we automatically pick the best among all possible orders for multiplying $n$ given matrices? Exhaustively examining all binary trees is impractical: There are $C(n) = \frac{1}{n} \binom{2n-2}{n-1} \approx \frac{4^n}{n\sqrt{n}}$ such trees ($C(n)$ is called the Catalan number of $n$). Naturally enough, dynamic programming is the answer.

Suppose that the matrices are $A_1 \times A_2 \times \cdots \times A_n$, with dimensions, respectively, $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$. Define a subproblem (remember, this is the most crucial and nontrivial step in the design of a dynamic programming algorithm the rest is usually automatic) to be to multiply the matrices $A_i \times \cdots \times A_j$, and let $M(i,j)$ be the optimum number of multiplications for doing so. Naturally, $M(i,i) = 0$, since it takes no effort to multiply a chain consisting of one matrix.

The recursive equation is

$$M(i,j) = \min_{i \leq k < j} \left[ M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j \right].$$

Naturally, $M(i,i) = 0$, since it takes no effort to multiply a chain consisting of the $i$th matrix. This equation defines the program and its complexity $-O(n^3)$.

for $i := 1$ to $n$ do $M(i,i) := 0$
  for $d := 1$ to $n - 1$ do
    for $i := 1$ to $n - d$ do
      $\{ j := i + d, M(i,j) = \infty, \text{best}(i,j) := \text{nil} \}$
      for $k := i$ to $j - 1$ do
        if $M(i,j) > M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$ then
          $\{ M(i,j) := M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j, \text{best}(i,j) := k \}$

As usual, improvements are possible (in this case, down to $O(n \log n)$).

Run this algorithm in the simple example of four matrices given to verify that the claimed order is the best!

**Optimum binary search trees.** Suppose that you know the frequency with which keywords appear in programs in a language:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>5%</td>
</tr>
<tr>
<td>do</td>
<td>40%</td>
</tr>
<tr>
<td>else</td>
<td>8%</td>
</tr>
<tr>
<td>end</td>
<td>4%</td>
</tr>
<tr>
<td>if</td>
<td>10%</td>
</tr>
<tr>
<td>then</td>
<td>10%</td>
</tr>
<tr>
<td>while</td>
<td>23%</td>
</tr>
</tbody>
</table>
We want to organize them in a binary search tree, such that the keyword in the root is lexicographically bigger than all keywords in its left subtree and smaller than all keywords in its right subtree (and the same for all other nodes) as shown below:

![Binary search tree diagram](image)

Figure 1: Binary search tree

The cost of this tree is 2.42—that is to say, the average keyword finds its position after 2.42 comparisons (1 comparison with probability 4% for 'end', 2 comparisons with probability 40 + 10 = 50% for 'do' and 'then', and so on. The optimum binary search tree is shown below; it uses 2.18 comparisons on the average, and can be found by dynamic programming.

Let \( p_i \) be the probability of the \( i \)th keyword, and define \( P_{ij} = \sum_{k=i}^{j} p_k \). Once more, let \( T(i, j) \) be the average number of comparisons in the optimum tree for keywords \( i \) through \( j \). It is easy to see that

\[
T(i, j) = \min_{r \leq j} [T(i, r - 1) + T(r + 1, j) + P_{ij}].
\]

Here \( r \) is the root of the optimum tree (to be determined by minimization). The equation states that any keyword in the range \( i \ldots j \) would cost us one for a comparison with the root, and, if it is not the root, \( T(i, r - 1) + T(r + 1, j) \) for the comparisons performed in the left or right subtrees. As always, the program is easy to write now (the initialization here is \( T(i,i - 1) = 0 \)).

2. Transitive Closure and the TSP

The following is an important problem: We are given a Boolean \( n \times n \) matrix \( G \)—or, equivalently, the adjacency matrix of a directed graph \( G = (V,E) \). We wish to determine for all \( i,j \in V \) whether there is a path from \( i \) to \( j \).

\[
G = \begin{bmatrix}
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
\end{bmatrix} \quad T(G) = \begin{bmatrix}
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 1 \\
\end{bmatrix}
\]
The matrix—or graph—$T(G)$ that contains the answers to all of these questions is called the (reflexive) transitive closure of $G$ (see above; notice that it contains all self-loops $(i, i)$). It is best computed via a dynamic programming technique of quite broad applicability.

The subproblem definition is here quite subtle. We seek all paths from $i$ to $j$. Such a path may or may not use intermediate nodes. Those that do not are the edge $(i, j)$, if it exists, and the self-loop $(i, i)$ if $i = j$. All other paths must use intermediate nodes. We define a subproblem by restricting the available pool of intermediate nodes. In particular, let $T^k(i, j)$ be true if and only if there is a path from $i$ to $j$ using intermediate nodes from among $1, 2, \ldots, k$ only. That is, we arbitrarily order the nodes of the graph, and allow more and more nodes as intermediate nodes in the path. The recursive equation is simple to write:

$$T^k(i, j) := T^{k-1}(i, j) \lor [T^{k-1}(i, k) \land T^{k-1}(k, j)].$$

It expresses this observation: Suppose that we have computed the paths that use intermediate nodes up to $k - 1$, and we wish now to also allow $k$. Consider paths from $i$ to $j$. There are two cases: Either such a path uses $k$, or it does not. If it does not, then $T^{k-1}(i, j)$ must be true. Otherwise, there must be a path from $i$ to $j$ going through $k$ just once. Thus, there must be a path from $i$ to $k$ using intermediate nodes up to $k - 1$, and a path from $k$ to $j$ using intermediate nodes up to $k - 1$.

The algorithm (known as the Floyd-Warshall algorithm), is now easy to write:

for $i, j := 1$ to $n$ do \{$T(i, j) := G(i, j)$ or $i = j$\}
for $k := 1$ to $n$
for $i, j := 1$ to $n$

$T(i, j) := T(i, j) \lor [T(i, k) \land T(k, j)]$

Notice that we have omitted the superscripts $k$. This results in savings in storage (since we only need the previous "layer" of $T$'s to compute the next). This may result in using, for example, in the computation of $T^3(5, 2)$ the value $T^3(3, 2)$ instead of the correct $T^2(3, 2)$ — but this does not result in mistakes, only in more "streamlined" computation of the transitive closure. The complexity is, of course $O(n^3)$.

Suppose now that in the above program you change the initialization to $T(i, j) := d(i, j)$ and $T(i, j) := 0$ if $i = j$, and in the loop we change $\lor$ to $\min$ and $\land$ to $+$. We get an $O(n^3)$ all-pairs shortest paths algorithm.

A similar manoeuvre yields an algorithm for transforming any finite automaton to the corresponding regular expression.

The traveling salesman problem. Suppose that you are given $n$ cities and the distances $d_{ij}$ between any two cities; you wish to find the shortest tour that takes you from your home city to all cities and back.

Naturally, the TSP can be solved in time $O(n!)$, by enumerating all tours — but this is very impractical. Since the TSP is one of the NP-complete problems, we have little hope of developing a polynomial-time algorithm for it. Dynamic programming gives an algorithm of complexity $O(n^2 2^n)$ — exponential, but much faster than $n!$. The recursive equation is similar to the one for the transitive closure: The main difference between the two problems is that in the transitive closure intermediate nodes are optional, while in the TSP they are mandatory.

We define the following subproblem: Let $S$ be a subset of the cities containing 1 and at least one other city, and let $j$ be a city in $S$ other than one. Define $C(S, j)$ to be the shortest path that starts from 1, visits all nodes in $S$, and ends up in $j$. The program now writes itself:

for all $j$ do $C(\{1\}, j) := d_{1j}$
for $s := 3$ to $n$ do (the size of the subsets considered this round)

for all subsets $S$ of $\{1, \ldots, n\}$ of size $n$ and containing 1 do

for all $j \in S, j \neq 1$
do

$\{C(S, j) := \min_{i, i \neq j, i \in S} [C(S - \{j\}, i) + d_{ij}]\}$
opt$:= \min_{j \neq 1} [C(\{1, 2, \ldots, n\}, j) + d_{1j}]$.

As always, we can also recover the optimum tour by remembering the $i$'s that achieve the minima. The complexity is $O(n^2 2^n)$: The table has $n^2 n$ entries (one per set and city), and it takes about $n$ time to fill each entry.

Dynamic Programming in Trees

Almost all NP-complete problems are easy to solve on trees. Let us take, for example, the independent set problem: You are given a graph, and you wish to find as many nodes as possible that have no edges going between them. It is immensely difficult to solve and approximate. Using dynamic programming, it can be solved in linear time when the graph is a tree.

Dynamic programming on trees usually proceeds as follows: Root the tree at an arbitrary node. Now each node defines a subtree (the one hanging from it). Dynamic programming proceeds, as always, from smaller to larger subproblems — that is to say, bottom-up in the rooted tree. Suppose that we know the size of the largest independent set of all subtrees below a node $i$. What is the maximum independent set in the subtree hanging from $i$? Two cases: either $i$ is in it, or it is not. If it is, then the maximum independent set consists of $i$, plus the
union of the maximum independent sets of the subtrees of the grandchildren of \( i \). If not, then it is simply the union of the maximum independent sets of the subtrees of the children of \( i \). The recursive equation is now easy to write:

\[
I(i) := \max \{ \sum_{j \text{ child of } i} I(j), 1 + \sum_{j \text{ grandchild of } i} I(j) \}.
\]

The complexity is linear: For each node, you only look at its children and its grandchildren; hence, each node will be looked at a constant number of times (when its maximum independent set is being computed, and again when its parent’s and its grandparent’s).