Dynamic Programming

So far in this book we have seen some elegant principles for the design of algorithms: divide-and-conquer, graph exploration, greedy choice. These techniques do work for some problems, but they are certainly specialized, they are delicate delicate tools. We now turn to the two *sledgehammers* of the algorithms craft, *dynamic programming* and *linear programming*. They are both techniques of very broad applicability which can be invoked when more specialized methods fail; but this generality often comes with a reduction in efficiency.

1 Shortest paths in dags, revisited

At the conclusion of our study of shortest paths, we observed that the problem is especially easy in directed acyclic graphs (dags). Let's recapitulate this case, because it lies at the heart of dynamic programming.

The special distinguishing feature of a dag is that its nodes can be *linearized*, that is, they can be arranged on a line so that all edges go from left to right (Figure 1.1). To see why this helps with shortest paths, suppose we want to figure out distances from node s to the other nodes. For concreteness, let's focus on node D. The only way to get to it is through its predecessors, B or C; so to find the shortest path to D, we need only compare these two routes:

$$\text{dist}(D) = \min\{\text{dist}(B) + 1, \text{dist}(C) + 3\}.$$

A similar relation can be written for every node. If we compute these dist values in the left-to-right order of Figure 1.1, we can always be sure that by the time we get to a node v, we already have all information we need to compute dist(v). We are therefore able to compute all distances in a single pass:

```
initialize all dist $(\cdot)$ values to $\infty$
dist $(s) = 0$
for each $v \in V\backslash\{s\}$, in topological order:
    dist $(v) = \min_{(u,v)\in E}\{$dist $(u) + l(u,v)\}$
```

Notice that this algorithm is solving a collection of *subproblems*, $\{\text{dist}(u) : u \in V\}$. We start with the smallest of them, dist(s), since we immediately know its answer to be zero. We then proceed with progressively "larger" subproblems (distances to vertices further and further to the right). That is, we think of a subproblem as large if it is far along in the linearization, that is, if we need to have solved a lot of other subproblems before we can get to it.

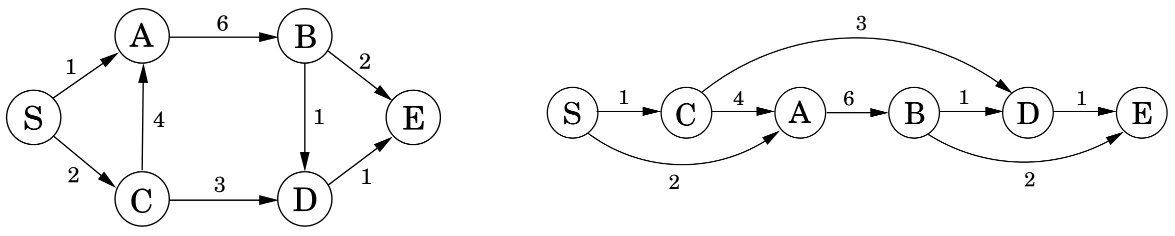
This is a very general technique. At each node, we compute some function of the values of the node's predecessors. It so happens that our particular function is a minimum of sums, but we could just as well make it a *maximum*, in which case we would get *longest* paths in the dag. Or we could use a product instead of a sum inside the brackets, in which case we would end up computing the path with the smallest product of edge lengths.

Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one-by-one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them are solved. In dynamic programming we are not given a dag; the dag is *implicit*. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems:

If subproblem B needs the answer to subproblem A to be solved, then there is a (conceptual) edge from A to B. And in this case, A is thought of a a smaller subproblem than B — and it will always be smaller, in an obvious sense.

But it's time we saw an example:

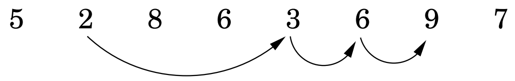
Figure 1.1 A dag and its topological ordering.



2 Longest increasing subsequences

[Connections: statistical tests for pseudorandomness; card games.]

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1 \cdots a_n$. A *subsequence* is any subset of these numbers taken in order, something of the form $a_{i_1} a_{i_2} \cdots a_{i_k}$ where $1 \le i_1 < i_2 < \cdots < i_k \le n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$:



In this little example, the arrows above denote transitions between consecutive elements in the optimal solution. More generally, to better understand the solution space, let's create a graph of *all* permissible transitions: establish a node i for each element a_i, and add directed edges (i, j) whenever it is possible for a_i, a_j to be consecutive elements in an increasing subsequence, that is, whenever $i < j$ and $a_i < a_j$ (Figure 2.1).

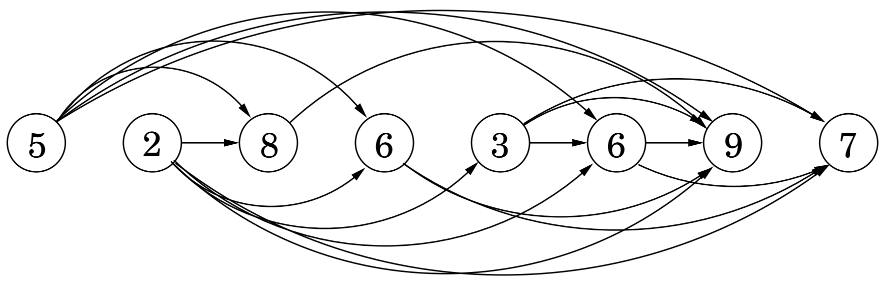
Notice that (1) this graph $G = (V, E)$ is a dag, since it respects the linear order $1, 2, \ldots, n$, and (2) there is a one-to-one correspondence between increasing subsequences and paths in this dag. Therefore, our goal is simply to find the longest path in the dag!

Here is the algorithm:

```
for $j = 1, 2, \ldots, n$:
    $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$
return $\max_j L(j)$
```

$L(j)$ is the length of the longest path — the longest increasing subsequence — ending at j (plus one, since strictly speaking we need to count nodes on the path, not edges). By reasoning in the same way as we did for shortest paths, we see that any path to node j must pass through one of its predecessors, and therefore $L(j)$ is one plus the maximum $L(\cdot)$ value of

Figure 2.1 The dag of increasing subsequences.



these predecessors. If there are no edges into j, we take the maximum over the empty set, zero. And the final answer is the *largest* $L(j)$, since any ending position is allowed.

This is dynamic programming. In order to solve our original problem, we have defined a collection of subproblems $\{L(j) : 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> (*) There is an ordering on the subproblems, and a relation which shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems which appear earlier in the ordering.

In our case, each subproblem is solved using the relation

$$L(j) \;=\; 1 + \max\{L(i) : (i,j) \in E\},$$

an expression which involves only smaller subproblems. How long does this step take? It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time (can you see how?), is handy. The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$. This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order. Thus the dynamic programming solution is both simple and efficient.[1]

There is one last issue to be cleared up: the L-values only tell us the length of the optimal subsequence, so how do we recover the subsequence itself? This is easy. While computing $L(j)$, we should also note down $\text{prev}(j)$, the predecessor through which the longest path to j was realized. The optimal subsequence can then be reconstructed by following these backpointers.

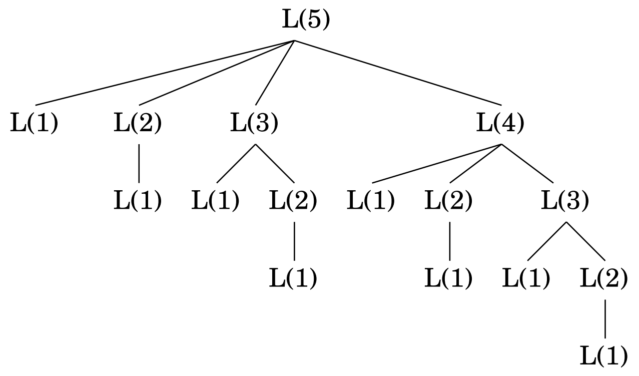
3 Edit distance

When a spell checker encounters a possible misspelling, it looks up its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?

A very natural measure of distance between strings x and y is the minimum number of *edits* — insertions, deletions, and substitutions of characters — needed to transform x into y.

[1]However, if one is willing to forgo some simplicity, a faster $O(n \log n)$ algorithm does exist (see Exercise).

Figure 2.2 The recursion tree for a sorted sequence of length five.



Recursion? No, thanks

The formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?
Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains all edges (i, j) for all $i < j$ — that is, the given sequence of numbers $a_1, a_2, \dots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \dots, L(j-1)\}.$$

Figure 2.2 unravels some of the initial stages of recursion for a small instance ($n = 5$). The key point to notice is that the same subproblems get solved over and over again!
In contrast, recursion was a perfect fit for divide-and-conquer The reason is that, in divide-and-conquer, we decompose a problem into independent, disjoint subproblems: the hierarchy of the problems is a tree (recall Figure 2.???). In contrast, in dynamic programming the subproblems we invoke have many common suproblems; the hierarchy of subproblems is, as we know, a dag, with a potentially exponential number of paths (recursive calls).

For instance, the edit distance between `to` and `fro` is 2:

$$\texttt{to} \longrightarrow \texttt{fo} \text{ (substitution)} \longrightarrow \texttt{fro} \text{ (insertion)}$$

(and clearly one edit isn't enough to do the job). Notice that edit distance is *symmetric*, in the sense that it doesn't depend on the order of its two arguments.

Let's figure out how to compute this distance.

A dynamic programming solution

When solving a problem by dynamic programming, the most crucial question is, *what are the subproblems?* As long as they are chosen so as to have the property (*) we discussed earlier, it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.

Programming?

The origin of the term *dynamic programming* has very little to do with writing code. It was first coined by Richard Bellman in the 1950s, a time when computer programming was an esoteric activity practiced by so few people as to not even merit a name. Back then "programming" meant "planning", and "dynamic programming" was conceived to optimally plan multi-stage processes. The dag of Figure 2.1 can be thought of as describing the possible ways in which such a process can evolve: each node denotes a state, the leftmost node is the starting point, and the edges leaving a state represent possible actions, leading to different states in the next unit of time.
The etymology of *linear programming*, the subject of the next chapter, is similar.

Our goal is to find the edit distance of two strings, say `EXPONENTIAL` and `POLYNOMIAL`. What is a good subproblem? Well, it should go part of the way towards converting one string to the other; so how about looking at the smallest number of edits needed to transform some *prefix* of the first string, such as `EXPO`, into some *prefix* of the second, say `POL`?

This is certainly plausible, but we also need to express large subproblems in terms of smaller ones. Let's think.

The edit distance between `EXPO` and `POL`, call it $E(4,3)$, is the number of edits needed to convert the first string into the second one. There are many ways to perform this transformation. We could start at the left-hand side of `EXPO` and systematically move right, gradually converting to `POL` as we go along:

`EXPO` – `XPO` (delete `E`) – `PO` (delete `X`) – `POL` (insert `L`).

Or we could move right-to-left, so that the next edit is always at the rightmost unmatched position of the source string:

`EXPO` – `EXPOL` (insert `L`) – `EPOL` (delete `X`) – `POL` (delete `E`).

Or equally, middle-out. What should we do?

Right-to-left seems the best strategy: if we can at least partially align the rightmost characters of the two strings, we should then be able to fall back on smaller subproblems like `EXP` $\rightarrow$`POL` or `EXP` $\rightarrow$`PO`. Now, given that the first edit will be at right end of `EXPO`, there are only three things it can be:

(a) *insert* `L`; (b) *delete* `O`; (c) *substitute* `O`$\rightarrow$`L`.

In the first case, the remaining edits must convert `EXPO` to `PO`, which we would call subproblem $E(4,2)$. In case (b), the remaining subproblem is to convert `EXP` to `POL`, $E(3,3)$. And in case (c), what's left is `EXP` $\rightarrow$`PO`, $E(3,2)$. In short, we have expressed $E(4,3)$ as one edit, followed one of the *smaller* subproblems $E(4,2), E(3,3), E(3,2)$. We don't know which one, so we just try them all and take the best: $E(4,3) = 1 + \min\{E(4,2), E(3,3), E(3,2)\}$.

More generally, to compute the edit distance between two words, word x with m letters and word y with n, we define $E(i,j)$ to be the edit distance between the prefixes $x[1\ldots i]$ and $y[1\ldots j]$. Obviously, our final objective is to compute $E(m,n)$.

To figure out $E(i,j)$, let's look at two cases. If the i^{th} letter of x differs from the j^{th} letter of y, the first operation in the transformation must be one of the following

- *delete* $x[i]$, in which case $E(i,j) = E(i-1,j)+1$; or
- *insert* $y[j]$, implying $E(i,j) = E(i,j-1)+1$; or
- *substitute* $x[i] \to y[j]$, implying $E(i,j) = E(i-1,j-1)+1$.

On the other hand, if the end characters of the two prefixes match, the first two options above remain legitimate, but for the third there is no increase in edit distance. It becomes

- $E(i,j) = E(i-1,j-1)$.

Therefore, we can calculate $E(i,j)$ as the minimum of these three possibilities! Specifically, we can express $E(i,j)$ in terms of the smaller subproblems $E(i-1,j-1), E(i-1,j)$, and $E(i,j-1)$.

We are almost done. There just remain the "base cases" of the dynamic programming, the very smallest subproblems. In the present situation, these are $E(0,\cdot)$ and $E(\cdot,0)$, both of which are easily solved. $E(0,j)$ is the fastest way to convert the 0-length prefix of x, namely the empty string, to the first j letters of y: clearly, j insertions. And similarly, $E(i,0)=i$, for i deletions.

At this point, the algorithm for edit distance basically writes itself. For convenience, let `diff` (i,j) be 1 if the i^{th} letter of x differs from the j^{th} letter of y, 0 otherwise.

```
for all $i,j$:   $E(i,0) = i$ and $E(0,j) = j$
for $i = 1,2,\ldots,m$:
    for $j = 1,2,\ldots,n$:
        $E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+$ diff $(i,j)\}$
return $E(m,n)$
```

The collection of subproblems forms a two-dimensional table $E(i,j)$, and this procedure fills in the table column by column, top to bottom, and left to right (Figure 3.1). Actually, there are many valid orderings: we just need to make sure that we always get to $(i-1,j-1),(i-1,j),(i,j-1)$ before (i,j). For instance, we could fill the table row by row, or in diagonals, incrementing $i+j$. Each entry takes constant time to fill in, so the overall running time is just the size of the table, $O(mn)$.

And in our example, the edit distance turns out to be 7:

```
EXPONENTIAL   --  EXPONENIAL   --  EXPONEMIAL   --  EXPONOMIAL   --
EXPOYNOMIAL   --  EXPOLYNOMIAL   --  EPOLYNOMIAL   --  POLYNOMIAL
```

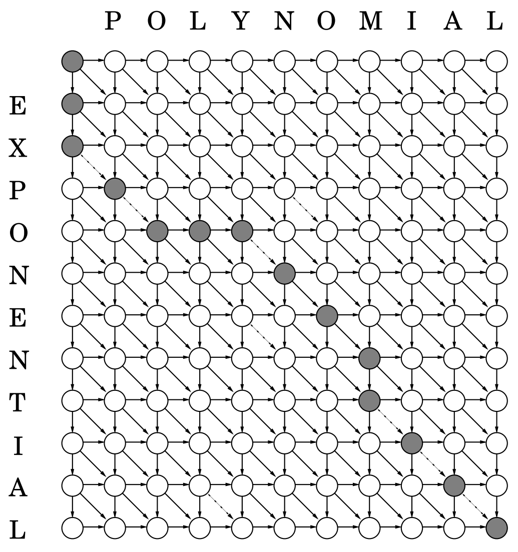
The underlying dag

Every dynamic program has an underlying dag structure: think of each node as representing a subproblem, and each edge as a precedence constraint on the order in which the subproblems can be tackled. Having nodes $i_1,\ldots,i_k$ point to j means "subproblem j can be solved once the answers to $i_1,\ldots,i_k$ are known."

In our present edit distance application, the nodes of the underlying dag correspond to subproblems, or equivalently, to positions (i,j) in the table. Its edges are the precedence constraints, of the form $(i-1,j)\to(i,j)$, $(i,j-1)\to(i,j)$, and $(i-1,j-1)\to(i,j)$ (Figure 3.1).

Figure 3.1 The distance between EXPONENTIAL and POLYNOMIAL is 7. *Left:* The final table of values found by dynamic programming. *Right:* The underlying dag, and a path of length 7.

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	4	2	3	4	5	5	6	7	8	9
N	5	5	3	4	5	4	5	6	7	8	9
E	6	6	4	4	5	5	5	6	7	8	9
N	7	7	5	5	6	5	6	6	7	8	9
T	8	8	6	6	6	6	6	7	7	8	9
I	9	9	7	7	7	7	7	7	7	8	9
A	10	10	8	8	8	8	8	8	8	7	8
L	11	11	9	8	9	9	9	9	9	8	7



Of mice and men

Discussion of genomics and the extra tricks used in BLAST

In fact, we can take things a little further and put weights on the edges so that the edit distances are given by shortest paths in the dag! To see this, set all edge lengths to one, except for $\{(i-1, j-1) \rightarrow (i, j) : x[i] = y[j]\}$ (shown dotted in the figure), whose length is zero. The final answer is then simply the distance between nodes $s = (0,0)$ and $t = (m,n)$. One possible shortest path is shown, the one that yields the alignment we found earlier. On this path, each move down is a deletion, each move right is an insertion, and each diagonal move is either a match or a substitution.

By altering the weights on this dag, we can allow generalized forms of edit distance, in which insertions, deletions and substitutions have different associated costs.

4 Knapsack

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or "knapsack") will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$. What's the most valuable combination of items he can fit into his bag?[2]

For instance, take $W = 10$ and

[2]If this application seems frivolous, replace "weight" with "CPU time" and "only W pounds can be taken" with "only W units of CPU time are available". Or bandwidth instead of CPU time, etc. The knapsack problem generalizes a wide variety of resource-constrained selection tasks.

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

There are two versions of this problem. If there are unlimited quantities of each item available, the optimal choice is to pick item 1 and two of item 4 (total: \$ 48). On the other hand, if there is one of each item (the burglar has broken in an art gallery, say), then the optimal knapsack contains items 1 and 3 (total: \$46).

As we shall see a few chapters down the line, neither version of this problem is likely to have a polynomial-time algorithm. However, using dynamic programming they can both be solved in $O(nW)$ time, which is reasonable when W is small, but is not polynomial since the input size is proportional to $\log W$, not W.

Let's start with the version which allows repetition. As always, the main question in dynamic programming is, what are the subproblems? Well, in this case we can shrink the original problem in two ways: we can either look at smaller knapsack capacities $w \leq W$, or we can look at fewer items (for instance, items $1, 2, \ldots, j$, for $j \leq n$). It usually takes a little experimentation to figure out exactly what works.

The first restriction calls for smaller capacities. Accordingly, define

$$K(w) = \text{maximum value achievable with a knapsack of capacity } w.$$

Can we express this in terms of smaller subproblems? Well, if the optimal solution to $K(w)$ includes item i, then removing this item from the knapsack should yield an optimal solution to $K(w - w_i)$. Of course, we don't know i, so let's try all possibilities:

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\},$$

where as usual our convention is that the maximum over an empty set is zero. We're done! The algorithm now writes itself, and it is characteristically simple and elegant.

```
$K(0) = 0$
for $w = 1$ to $W$:
    $K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
return $K(W)$
```

This algorithm fills in a one-dimensional table of length $W + 1$, in left-to-right order. Each entry can take up to $O(n)$ time to compute, so the overall running time is $O(nW)$. As always, we can also construct the underlying dag. Doing so gives us a startling insight: this particular variant of knapsack is nothing other than longest path in a dag! (Check!)
The dag of this dynamic program is also very simple: The vertices are all integers between 0 and W, and there is an edge between w and w' whenever $w' = w + w_i$ for some $i \leq n$.

Onto the second variant: what if repetitions are not allowed? Our earlier subproblems now become completely useless. Because, knowing that the value $K(W - w_n)$ is very high doesn't help us, because we don't know whether or not item n already got used up in this partial solution. We must thereore refine our concept of a subproblem. Our subproblems need to carry additional information about the items being used. We add a second parameter, $0 \leq j \leq n$:

$$K(w, j) = \text{maximum value achievable using a knapsack of capacity } w \text{ and items } 1, \ldots, j.$$

The answer we seek is $K(W, n)$.

How can we express a subproblem $K(w, j)$ in terms of smaller subproblems? Quite simple: either item j is needed to achieve the optimal value, or it isn't needed:

$$K(w, j) = \max\{K(w - w_j, j - 1), K(w, j - 1)\}.$$

(The first case is invoked only if $w_j \leq w$.) In other words, we can express $K(w, j)$ in terms of subproblems $K(\cdot, j - 1)$.

The algorithm then consists of filling out a two-dimensional table, with $W + 1$ rows and $n + 1$ columns. Each table entry takes just constant time, so even though the table is much larger than in the previous case, the running time remains the same, $O(nW)$. Here's the code.

```
Initialize  all  $K(0, j) = 0$  and  all  $K(w, 0) = 0$
for  $j = 1$  to  $n$:
    for  $w = 1$  to  $W$:
        if  $w_j > w$:   $K(w, j) = K(w, j - 1)$
        else:   $K(w, j) = \max\{K(w, j - 1), K(w - w_j, j - 1)\}$
return  $K(W, n)$
```

5 Chain matrix multiplication

Suppose that we want to multiply four matrices, $A \times B \times C \times D$, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively. This will involve iteratively multiplying two matrices at a time. Matrix multiplication is not *commutative* (in general, $A \times B \neq B \times A$), but it is *associative*, which means for instance that $A \times (B \times C) = (A \times B) \times C$. Thus we can compute our product of four matrices in many different ways, depending on how we parenthesize it. Are some of these better than others?

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications, to a good enough approximation. Using this formula, let's compare several different ways of evaluating $A \times B \times C \times D$:

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

As you can see, the order of multiplications makes a big difference in the final running time! Moreover, the natural *greedy* approach, to always perform the cheapest matrix multiplication available, leads to the second parenthesization shown here, and is therefore a failure.

How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \cdots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively? The first thing to notice is that a particular parenthesization can be represented very naturally by a binary tree in which the individual matrices correspond to the leaves, the root is the final product, and interior nodes are intermediate products (Figure 5.1). The possible orders in which to do the multiplication correspond to the various binary trees with n leaves, whose number is exponential in n (check!). We certainly cannot try each tree, and with brute force thus ruled out, we turn to dynamic programming.

The binary trees of Figure 5.1 are suggestive: for a tree to be optimal, its subtrees must also be optimal. What are the subproblems corresponding to subtrees? They are products of the form $A_i \times A_{i+1} \times \cdots \times A_j$. Let's see if this works. For $i \leq j$, we define

$$C(i,j) = \text{minimum cost of multiplying } A_i \times A_{i+1} \times \cdots \times A_j.$$

The size of this subproblem is the number of matrix multiplications, $|j - i|$. The smallest subproblem is when $i = j$, in which case there's nothing to multiply, so $C(i,i) = 0$. For $j > i$, consider the optimal subtree for $C(i,j)$. The first branch in this subtree, the one at the top, will split the product in two pieces, of the form $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$, for some k between i and j. The cost of the subtree is then the cost of these two partial products, plus the cost of combining them: $C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j$. And we just need to find the splitting point k for which this is smallest:

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

We are ready to code! In the following, the variable s denotes subproblem size.

```
for  $i = 1$  to  $n$:   $C(i,i) = 0$
for  $s = 1$  to  $n - 1$:
   for  $i = 1$  to  $n - s$:
      $j = i + s$
      $C(i,j) = \min\{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j : i \leq k < j\}$
return  $C(1,n)$.
```

The subproblems constitute a two-dimensional table, each of whose entries can take $O(n)$ time to compute. The overall running is then $O(n^3)$.

6 Shortest Paths

We started this chapter with the most elementary, stripped-down dynamic programming algorithm, finding shortest paths in dags. Not surprisingly, it turns out that more elaborate dynamic programming algorithms help solve quite sophisticated shortest-path problems.

Shortest Reliable Paths

Life is complicated, and abstractions such as graphs, edge lengths, and shortest paths rarely capture the whole truth. For example, in a communications network, even though edge lengths may faithfully reflect transmission delays, there may be other considerations involved in choosing a path. For example, each extra edge in the path may be an extra "hop" frought with uncertainties and dangers of packet loss. For this reason, we may avoid paths that have too many edges.

For example, in the Graph of Figure **??** the shortest path from s to t has four edges, while another path, a little longer but with two edges exists. If four edges translate to prohibitive unreliability, we may have to choose the latter.

Suppose then that we are given a graph G with lengths on the edges, two nodes s and t, and an integer Q. We want the shortest path from s to t *with* Q *or fewer edges*.

One could try to adapt Dijkstra's algorithm to this new task. The problem is that Dijkstra's algorithm focuses on the `dist` information, the length of the shortest path to the current vertex. It does not "remember" the number of hops in that path, presently a crucial piece of information.

In dynamic programming, the trick is to choose subproblems so that all vital information is "remembered" and carried forward. In this case, let us define, for each vertex v and each integer $i \leq Q$, `dist` (v, i) to be *the length of the shortest path from* s *to* v *that has* i. Initially, `dist` $(v, 0)$ is ∞ for all vertices except s, for which it is 0. The equation is, naturally enough,

$$\texttt{dist}(v, i) = \min_{(u,v) \in E} [\texttt{dist}(u, i-1) + \ell(u, v)].$$

Need we say more?

All-pairs Shortest Paths

What if we want to find the shortest path between u and v *all* pairs of vertices? One way would be to execute the Bellman-Ford algorithm (since there may be negative edges) $|V|$ times, once for each starting node u, for a total of $O(|V|^2|E|)$.

A better algorithm, called *the Floyd-Warshall algorithm*, takes time $O(|V|^3)$, and is based on dynamic programming.

We want to find the distances between all pairs of vertices in a graph. What is a subproblem of this? Solving the problem for more and more pairs or starting points is not a good idea — it leads to the $O(|V|^2|E|)$ algorithm.

One idea comes to mind: The shortest path $(u, w_1, w_2, \ldots, w_k, v)$ between u and v uses several intermediate nodes — possibly zero of them. When no intermedaie nodes are allowed, we know the answer: Direct edges, whenever they exist. What if we allow more and more intermediate nodes? This works, but it would be a $|V|$-fold repetition of the shortest reliable paths algorithm of the previous subsection, with $Q = |V| - 1$, for a total of $O(|V|^4)$ steps — even the $|V|$ repetitions of Bellman-Ford beats that.

Here is a more radical idea for building upon the direct edges solution: Suppose that we restrict the *set* I of vertices that can be used as intermediate nodes in a path. In the beginning I is empty, and only direct edges are allowed. Then we add to I the nodes of the graph, one by one, updating the shortest path lengths. When $I = V$, all vertices are allowed to be on all paths, and we have found the true shortest paths between all vertices of the graph!

To be more concrete, suppose that we have numbered the vertices in V as $\{1, 2, \ldots, n\}$, and let `dist` (i, j, k) denote the length of the shortest path from i to j using as intermediate nodes only nodes from the set $\{1, 2, \ldots, k\}$. Initially, `dist` $(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and it is ∞ otherwise.

What happens when we add a new node k to I? We must reexamine all pairs i and j and see whether using k as a stop makes the shortest path from i to j any shorter. But this is easy to check: A shortest path from i to j that uses k along with possibly other, lower-numbered intermediate nodes, goes through k just once (why? because we assume that there are no negative cycles). And we have already calculated the length of the shortest path from i to k and from k to j using only lower-numbered vertices. That is, we just need to check see whether dist $(i, k+1, k)$ + dist $(k+1, j, k)$ is smaller than dist (i, j, k) — if so, dist (i, j, k) must be updated to get dist $(i, j, k+1)$.

Here is the Floyd-Warshall algorithm:

```
for $i = 1$ to $n$:
  for $j = 1$ to $n$:
    if $(i, j)$ is in $E$ then dist $(i, j, 0) = \ell(i, j)$
      else dist $(i, j, 0) = \infty$
for $k = 1$ to $n$:
  for $i = 1$ to $n$:
    for $j = 1$ to $n$:
     dist $(i, j, k) = \min\{$dist $(i, k, k-1) +$ dist $(k, j, k-1),$ dist $(i, j, k-1)\}$
```

As promised, it takes $O(|V|)^2$ time.

The Traveling Salesman Problem

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

Denote the cities by $1, \ldots, n$, the salesman's hometown being 1, and let $D = (d_{ij})$ be the matrix of intercity distances. The goal is to design a tour which starts and ends at 1, includes all other cities exactly once, and has minimum total length. Figure 6.1 shows a example involving five cities. Can you spot the optimal tour? Even in this tiny example, it is tricky for a human to find the solution; imagine what happens when hundreds of cities are involved.

It turns out this problem is also difficult for computers. In fact, the traveling salesman problem (TSP) is one of the most notorious computational tasks. There is a long history of attempts at solving it, a long saga of failures and partial successes, and along the way, major advances in algorithms and complexity theory. The most basic piece of bad news about the TSP, which we will better understand in a couple of chapters, is that it is highly unlikely to be solvable in polynomial time.

How long does it take, then? Well, the brute-force approach is to evaluate every possible tour and return the best one. Since there are $n!$ possibilities, this strategy takes time $O(n \cdot n!)$. We will now see that dynamic programming yields a much faster solution, though not a polynomial one.

What is the appropriate subproblem for the TSP? Subproblems refer to partial solutions, and in this case the most obvious partial solution is the initial portion of a tour. Suppose we have started at city 1, as required, have visited a few cities, and are now in city j. What information do we need in order to extend this partial tour? We certainly need to know j, since this will determine which cities are most convenient to visit next. And we also need to know all the cities visited so far, so that we don't repeat any of them. Here, then, is an appropriate subproblem.

> For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ which includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1, and ending at j.

Now, let's express $C(S, j)$ in terms of smaller subproblems. We need to start at 1, and end at j. What should we pick as the second-last city? It has to be some $i \in S$, which case the overall path length is the distance from 1 to i, namely $C(S - \{j\}, i)$, plus the length of the final edge, d_{ij}. We must pick the best such i:

$$C(S, j) = \min_{i \in S - \{j\}} C(S - \{j\}, i) + d_{ij}.$$

The subproblems are ordered by $|S|$. Here's the code.

```
for $j = 1$ to $n$:      $C(\{1, j\}, j) = d_{1j}$
for $s = 3$ to $n$:
    for all subsets $S \subseteq \{1, 2, \ldots, n\}$ of size $s$ and containing $1$:
        for all $j \in S, j \neq 1$:
            $C(S, j) = \min_{i \in S, i \neq j} C(S - \{j\}, i) + d_{ij}$
return $\min_{j \neq 1} C(\{1, \ldots, n\}, j) + d_{j1}$.
```

There are $2^n \cdot n$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 2^n)$.

7 Independent sets in trees

A subset of nodes $S \subset V$ is an *independent set* of graph $G = (V, E)$ if there are no edges between them. For instance, in Figure 7.1 the nodes $\{1, 5\}$ are an independent set, but $\{1, 4, 5\}$ are not, because of the edge between 4 and 5. The largest independent set is $\{2, 3, 6\}$.

Like several other problems we have seen in this chapter (knapsack, traveling salesman problem), finding the largest independent set in a graph is believed to be intractable. However, when the graph happens to be a *tree* the problem can be solved in linear time, using dynamic programming. And what are the appropriate subproblems? Already in the chain matrix multiplication problem we noticed that the layered structure of a tree provides a natural definition of a subproblem — as long as one node of the tree has been identified as a root.

Here's the algorithm: start by rooting the tree at any node r. Now, each node defines a subtree – the one hanging from it. This immediately suggests subproblems:

$$I(u) \;=\; \text{the size of the largest independent set of the subtree hanging from } u.$$

Our final goal is $I(r)$.

Dynamic programming proceeds as always from smaller subproblems to larger ones, that is to say, bottom-up in the rooted tree. Suppose we know the largest independent sets for all subtrees below a certain node u; in other words, suppose we know $I(w)$ for all descendants w

of u. How can we compute $I(u)$? Let's split the computation into two cases: any independent set either includes u or it doesn't (Figure 7.2).

$$I(u) \;=\; \max \left\{ 1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \quad \sum_{\text{children } w \text{ of } u} I(w) \right\}.$$

If the independent set includes u, then we get one point for it, but we aren't allowed to include the children of u – therefore we move onto the grandchildren. This is the first case in the formula. On the other hand, if we don't include u, then we don't get a point for it, but we can move on to its children.

The number of subproblems is exactly the number of vertices. With a little care, the running time can be made linear, $O(|V| + |E|)$.

Memoization

In dynamic programming, we write out a recursive formula which expresses large problems in terms of smaller ones, and then use it to fill out a table of solution values in a bottom-up manner, from smallest subproblem to largest.

The formula also suggests a recursive algorithm, but we saw earlier that naive recursion can be terribly inefficient, because it solves the same subproblems over and over again. What about a more intelligent recursive implementation, one that remembers its previous invocations and thereby avoids repeating them?

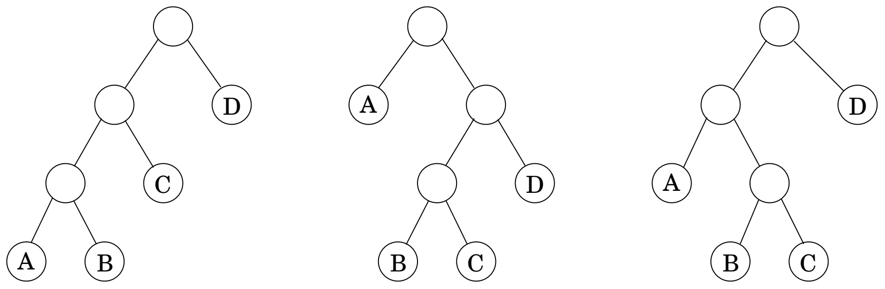
Such an algorithm would use hashing to store the values of $K(w)$ which have already been computed. At each recursive call $K(w)$, the algorithm would first check if the answer is already in the hash table, and proceed to its calculation only when it is not. This trick is called *memoization*:

```
knapsack( $W$)
declare a hash table $H$, initially empty,
containing values of $K(w)$ indexed by $w$
     ⋮
function knapsack($w$)
if $w$ is in $H$ return $K(w)$
else $K(w) = \max\{$knapsack $(w - w_i) + v_i : w_i \leq w\}$
insert $K(w)$ in $H$ with key $w$
return $K(w)$
```

Since this algorithm never repeats a subproblem, its running time is $O(nW)$, just like the dynamic program. However, the constant factor in this big-O is substantially larger because of the overhead of recursion.

In some cases, though, memoization pays off. Here's why: dynamic programming automatically solves every subproblem that could conceivably be needed, while memoization only ends up solving the ones that are actually used. For instance, suppose that W and all the weights w_i are multiples of 100. Then any subproblem $K(w)$ is useless unless 100 divides w. The memoized recursive algorithm will never look at these extraneous table entries.

Figure 5.1 (i) $((A \times B) \times C) \times D$; (ii) $A \times ((B \times C) \times D)$; (iii) $(A \times (B \times C)) \times D$.



On Time and Memory

The amount of time it takes to run a dynamic programming algorithm is easy to discern once you know the dag of the subproblems: *It is the total number of edges in the dag!* The reason is simple: All we do in dynamic programming is visit the nodes in linearized order, and at each node examine one-by-one the edges coming into it, typically doing constant work for each edge. This way, each edge is examined once.

But how much computer memory is required? There is no simple parameter of the dag that characterizes the memory requirements of the algorithm. But usually one can get away with much less memory than one word for every subproblem (vertex in the dag). The reason is that the space used to store the value of a subproblem can be released for reuse as soon as all larger subproblems that depend on this subproblem have been solved.

For example, in the Floyd-Warshall algorithm the value of dist (i, j, k) is not needed after we are done computing the dist $(\cdot, \cdot, k + 1)$ values. Therefore, one needs only two $|V| \times |V|$ arrays to store the dist values, one for odd values of k and one for even values. In other words, when computing dist (i, j, k) we overwrite the value of dist $(i, j, k - 2)$.

(And let us not forget that, as always in dynamic programming, we also need one more array, prev (i, j) storing the next to last vertex in the current shortest path from i to j, a value that must be updated with dist (i, j, k). We omit this mundane but crucial bookkeeping step from our dynamic programming algorithms.)

Can you see why the edit distance dag in (Figure 3.1) only needs memory equal to the length of the shorter of the two strings?

Figure 6.1 The optimal traveling salesman tour has length ten.

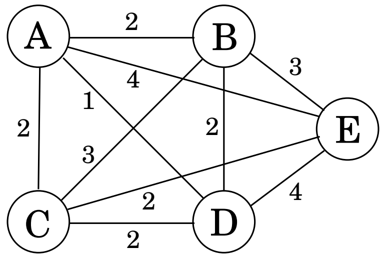


Figure 7.1 The largest independent set in this graph has size three.

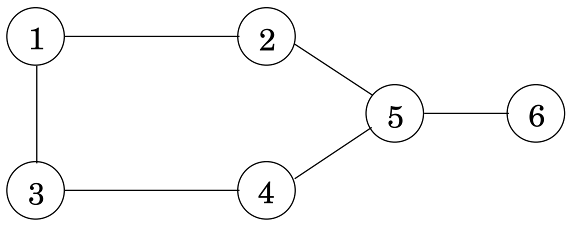


Figure 7.2 $I(u)$ is the size of the largest independent set of the subtree rooted at u. Two cases: either u is in this independent set, or it isn't.