Dynamic programming algorithms for combinatorial optimization generalize the strategy we studied for design and analysis of greedy algorithms. To build a proof of optimality, we first expand the proof structure for greedy algorithms to accommodate multiple first choices and multiple subproblems. We then reduce the high-level algorithm description to a standard recurrence form and find an efficient computation order for it.

1  A Running Example: Matrix Chain Multiplication

Let $A_1 \ldots A_N$ be a collection of matrices, where $A_i$ has size $p_i \times q_i$. Provided that these matrices are conformable ($q_i = p_{i+1}$ for every $i$), they can be multiplied. Multiplying an $n \times p$ and a $p \times m$ matrix requires, to a first approximation, $nmp$ operations. Because matrix multiplication is associative, any parenthesization of the product computation yields the same result; however, not all parenthesizations have the same cost.

Our problem is to find a parenthesization of the matrix chain that minimizes the total cost of all pairwise multiplications needed to compute its product.

2  A High-Level Algorithm and Its Correctness

Optimal parenthesization of matrix chains does not appear to have a good greedy algorithm. That is, no single first choice is necessarily consistent with optimality. We will therefore consider algorithms that make a collection of possible first choices and keep the one that yields the best answer.

As the first choice in our example, we will decide how to split the chain $A_1 \ldots A_N$ into two parts, $A_1 \ldots A_k$ and $A_{k+1} \ldots A_N$. To compute the product of the chain, we first compute product $\Pi_1$ from $A_1 \ldots A_k$ and $\Pi_2$ from $A_{k+1} \ldots A_N$, then compute the final product as $\Pi_1 \cdot \Pi_2$.

There are $N - 1$ locations at which we could split the chain $A_1 \ldots A_N$ into two nonempty parts. If we were to recursively compute the costs of the best solutions that make each of these $N - 1$ initial splits, we could improve our result by comparing these solutions and returning the best one overall.

Trying multiple first choices might be a good heuristic, but does it guarantee an optimal solution? To argue for optimality, we will extend the inductive “proof template” from our work with greedy algorithms to accommodate more than one first choice. Here’s roughly how the proof will look:

1. **To prove:** the proposed algorithm yields an optimal solution for problem instances of all sizes.

2. We will proceed by induction on the size of the problem instance $P$.

3. **Bas:** For the smallest possible $P$, we show directly that we obtain an optimal solution. (The smallest $P$ is typically empty or, if that is not allowed, contains only one element.)
4. **Ind**: Let \( P \) be given, and let \( c_1 \ldots c_n \) be the list of initial choices investigated by the algorithm for \( P \).

5. After making choice \( c_k, 1 \leq k \leq n \), show that we are left with \textit{one or more} strictly smaller subproblems \( P'_{k,1} \ldots P'_{k,m} \) of \( P \).

6. By the IH, applying our algorithm to each subproblem yields optimal subsolutions \( \Pi'_{k,1} \ldots \Pi'_{k,m} \).

7. We then show that combining the subsolutions with choice \( c_k \) yields a feasible solution \( \Pi_k \) to \( P \), and that this solution is optimal among all solutions making choice \( c_k \).

8. Finally, we argue that at least one choice in \( c_1 \ldots c_n \) is consistent with some globally optimal solution to \( P \), and so taking the best \( \Pi_k \) among \( 1 \leq k \leq n \) yields a global optimum. QED

Fleshing out this template requires that we provide a base case argument in point 3 (required, but usually easy), and, crucially, that we provide justifications for points 5 and 7-8. Here follows some advice on how to supply these essential parts of the proof.

8. **Show that at least one choice** \( c_k, 1 \leq k \leq n \), **is compatible with optimality**. In other words, show that there is always an optimal solution to \( P \) that makes one of the choices \( c_1 \ldots c_n \); hence, finding the best solution given each of these choices is guaranteed to discover a global optimum. This claim is the **complete choice property** for the algorithm.

   Here’s an example of this argument for our example problem. Suppose \( \Pi^* \) is an optimal solution to problem instance \( P \). At its top level, \( \Pi^* \) splits the matrices \( A_1 \ldots A_N \) into two groups \( A_1 \ldots A_k \) and \( A_{k+1} \ldots A_N \) that are multiplied together. Let \( c_k \) be the choice of splitting \( P \) after \( A_k \); then \( \Pi^* \) must make one of the choices \( c_1 \ldots c_{N-1} \). Hence, finding optimal solutions given each of these choices and keeping the best result overall is guaranteed to yield a global optimum. QED

   **Note**: much of the time, as in the above example, this part of the proof is trivial because we simply make \textit{every possible} first choice; hence, it is not hard to see that one such choice appears in some optimal solution. However, it may sometimes be possible to show that a more restricted set of first choices is guaranteed to produce an optimum. In the extreme case of greedy algorithms, one choice is always sufficient.

5 and 7a. **Show that for all first choices** \( c_k \), **making choice** \( c_k \) **leaves one or more strictly smaller versions of the original problem, each with no external constraints**. This is the **inductive structure property**, just as in the greedy case. Note that you need to prove the property for \textit{every} initial choice \( c_k \) and every subproblem induced by this choice; however, you may be able to write the proof just once if all the proofs would be similar.

   In our example, making first choice \( c_k \) leaves two subproblems \( P'_{k,\ell} \) and \( P'_{k,r} \), corresponding to the matrix chains \( A_1 \ldots A_k \) and \( A_{k+1} \ldots A_N \), respectively. Each of these two subproblems is strictly smaller than \( P \). Moreover, any solutions \( \Pi'_{k,\ell} \) and \( \Pi'_{k,r} \) to the subproblems may feasibly be combined by multiplying the resulting product matrices, so there are no external constraints imposed on the subproblems’ solutions. The above holds for all \( 1 \leq k \leq N - 1 \).

7b. **Show that for all first choices** \( c_k \), **combining choice** \( c_k \) **with optimal solutions to its induced subproblems yields a solution** \( \Pi_k \) **to** \( P \) **that is optimal among all**
solutions making choice $c_k$. This is the *optimal substructure property*, just as in the greedy case. Again, you need to prove the property for *every* first choice $c_k$. Each of these proofs can be done using the standard form of contradiction argument we saw in the greedy case.

In our example, suppose that making choice $c_k$ leaves subproblems $P'_{k,ℓ}$ and $P'_{k,r}$ with optimal solutions $Π'_{k,ℓ}$ and $Π'_{k,r}$. Combining these subsolutions with choice $c_k$ yields a parenthesization $Π_k$ for $P$.

Denote by $V(Π)$ the objective value associated with a solution $Π$. By the problem definition, we have that

$$V(Π_k) = V(Π'_{k,ℓ}) + V(Π'_{k,r}) + p_1q_kq_N.$$ 

Now suppose that $Π_k$ is *not* optimal for $P$ among all solutions making choice $c_k$. Let $Π^*_k$ be such an optimal solution. Because $Π^*_k$ makes the same initial choice as $Π_k$, we have that

$$V(Π^*_k) = V(Π'_{k,ℓ}) + V(Π'_{k,r}) + p_1q_kq_N$$

for some subsolutions $Π'_{k,ℓ}$ and $Π'_{k,r}$. But if $Π^*_k$ is better than $Π_k$, it must be that

$$V(Π'_{k,ℓ}) + V(Π'_{k,r}) > V(Π'_{k,ℓ}) + V(Π'_{k,r}),$$

which is not possible unless at least one of $Π'_{k,ℓ}$ or $Π'_{k,r}$ is better than $Π'_{k,ℓ}$ or $Π'_{k,r}$, respectively. But this contradicts the optimality of the $Π'$s. Conclude that $Π_k$ must in fact be optimal for $P$ among all solutions making choice $c_k$. QED

*Note:* as in the greedy case, this argument works without modification for any objective $V$ that can be decomposed into a sum or product of parts, one of which depends only on the initial choice $c_k$ and the others of which depend only on the induced subproblems.

### 3 Describing an Efficient Solution

We now know that our proposed algorithm is optimal, but we don’t yet have a fully specified implementation of it. We will now develop a *recurrence*, or formal recursive description of our algorithm, which will help us work out an efficient way to implement it. Along the way, we will borrow some notation from the theory of *polyhedral analysis*, which is used by compiler writers and hardware designers to efficiently implement nested loops and other recursive or iterative computations.

#### 3.1 Building a Recurrence

Our recursive algorithm solves a large number of subproblems of the full problem $P$. To formally describe the algorithm, we will develop a simple notation for subproblems and use our observations from the optimality proof to describe how a subproblem’s solution is built up from solutions to even smaller subproblems.

We will index the subproblems of a problem instance $P$ by a vector $z$ of one or more integer *coordinates*. For example, subproblem $z = [i, j]$ in our example seeks the optimal parenthesization of $A_i \ldots A_j$, where $1 \leq i \leq j \leq N$. The set of all subproblems induced by a problem instance $P$ is called the *domain* of that instance, which we will denote by $D$. Individual subproblems $z \in D$ are sometimes referred to as *points* because they are described by their coordinates.
Now that we can formally identify subproblems, we will show how to recursively compute the objective value \( V[z] \) of an optimal solution to a subproblem \( z \in D \). Informally, the computation is “for each initial choice, compute the value of an optimal solution that makes that choice, then keep the best solution overall.” The set of initial choices was part of our first, informal algorithm description, while the value of an optimal solution that makes this choice was computed in the proof of its substructure property.

Suppose there are \( C(z) \) possible initial choices for subproblem \( z \), and let \( V[z \mid k] \) be the value of an optimal solution given choice \( c_k \). Then, if the optimization is a minimization,

\[
V[z] = \min_{1 \leq k \leq C(z)} V[z \mid k].
\]

(For a maximization, replace the min by a max.) The expression for each term of the minimization is given by its decomposition into subproblems. The resulting equation is a recurrence for computing \( V[z] \).

For our example, let \( V[i, j] \) be the cost of an optimal parenthesization of \( A_i \ldots A_j \). We have that

\[
V[i, j] = \min_{i \leq k < j} V[i, k] + V[k + 1, j] + p_i q_k q_j.
\]

If you prefer, you can combine these two definitions, like this:

\[
V[i, j] = \min_{i \leq k < j} (V[i, k] + V[k + 1, j] + p_i q_k q_j).
\]

(In more complex recurrences, the terms for the value of each choice may have different mathematical formulas. Consult your optimal substructure proof for guidance.)

To compute any recurrence, we need it to “bottom out” at non-recursive base cases, that is points \( z \in D \) whose values \( V[z] \) are known a priori. In our example, \( V[i, i] = 0 \) for any \( i \), since there is no cost to compute a product of one matrix. Note that, as we descend the levels of the recursion, distances between \( i \) and \( j \) shrink, so we will eventually reach these base cases.

Finally, we need to identify a goal point \( z^* \) in the domain that represents the solution to our full problem instance. In this case, the point \( [1, N] \) corresponds to the full instance, so the value \( V[1, N] \) is the value of an optimal solution to the problem instance \( P \).

Note: while computing \( V[1, N] \) only gives us the objective value of an optimal solution, the solution itself is given by the choices made, i.e. the argmins of each minimization starting from the goal point and descending to the base cases. Each minimization indicates where to split a subchain of the original matrix chain. This relation between computing the optimal objective value and obtaining the actual solution holds for every recurrence.

### 3.2 Solving the Recurrence Efficiently

A recurrence gives a formula for the quantity we wish to compute, but it does not fully specify how to evaluate the formula. Naively, we could evaluate the recurrence “top-down” by generating the subproblems of the goal point, solving these recursively in turn, and then combining the solutions as the recurrence directs. However, a top-down procedure is grossly inefficient. Even if each initial choice generated only one subproblem, the resulting computation would have a branching factor...
equal to the number of initial choices, which is at least 2 for non-greedy algorithms. If the recursion depth is polynomial in the input size, the resulting algorithm requires time exponential in that size!

The problem with a top-down solution is that it winds up repeatedly solving the same subproblems. For example, \( V[1, 3] \) and \( V[2, 4] \) in our example problem both require computation of \( V[2, 3] \).

Fortunately, the polyhedral view of our recurrence suggests a better approach. To compute our goal value \( V[z^*] \), we may at worst need to evaluate the objective \( V \) at every point \( z \in \mathcal{D} \). Let’s define an evaluation order on \( \mathcal{D} \) that lets us solve each subproblem exactly once. As we solve each subproblem \( z \), we will store its value \( V[[z]] \) in a table so that we can look it up in \( O(1) \) time whenever we need it later. This “bottom-up” evaluation of a recurrence is called dynamic programming.

Not every evaluation order on \( \mathcal{D} \) is feasible. For example, we can’t compute \( V[1, 3] \) in our matrix problem until we know both \( V[1, 2] \) and \( V[2, 3] \). In general, each point \( z \in \mathcal{D} \) has one or more dependencies on smaller subproblems, which are specified by the recurrence itself. To realize a dynamic programming computation, we must find an evaluation order for \( \mathcal{D} \) that is consistent with these dependencies.

For our example, observe that \( V[i, j] \) depends on the values of subproblems \( V[r, s] \), where \( i \leq r \leq s \leq j \) and \( s - r < j - i \). We can therefore satisfy the recurrence’s dependencies by solving subproblems \( V[i, j] \) in increasing order of their difference \( d = j - i \). The following pseudocode shows one total ordering of subproblems that meets this criterion:

\[
\text{for } d = 1 \ldots N - 1 \\
\quad \text{for } i = 1 \ldots N - d \\
\quad \quad j \leftarrow i + d \\
\quad \quad V[i, j] \leftarrow \min_{i \leq k < j} (V[i, k] + V[k + 1, j] + p_i q_k q_j)
\]

\[
\text{return } V[1, N]
\]

In general, the cost of a dynamic programming algorithm is the domain size \( |\mathcal{D}| \) times the cost to compute each point \( V[z] \) given its dependencies. In this case, we have a two-dimensional domain with \( i \leq j \) running from 1 to \( N \), so the domain has \( \Theta(N^2) \) points. Each point performs a minimization over at most \( N \) initial choices and so requires worst-case time \( \Theta(N) \) to compute. Hence, the total cost of the algorithm is \( \Theta(N^3) \).

4 Advice for Your Homework

To develop a dynamic programming algorithm for class purposes, you should do all of the following:

1. Describe your set of initial choices.

2. Prove optimality of your proposed algorithm by addressing each of the three required properties:

   (a) complete choice

   (b) inductive structure

   (c) optimal substructure
Please clearly divide your argument into the above three pieces, with suitable headings. You need not write down the full inductive template, though it may help you to do so the first couple of times to make sure you understand your proof obligations. Feel free to abbreviate your optimal substructure proofs in the common case as for greedy algorithms.

3. Develop a recurrence to precisely describe your algorithm. You must provide all of the following:

(a) a definition of the domain (give the coordinates of a general subproblem, e.g. \([i, j]\), and explain what they mean);

(b) a formula for computing the value of a general subproblem from its dependencies;

(c) one or more base cases sufficient to ground the recurrence;

(d) identification of the goal point \(z^\ast\).

The hardest part of writing the recurrence is developing its formula, but you should have already done this work as part of your optimal substructure proof.

4. Give an evaluation order on \(D\) that satisfies all dependencies of the recurrence, evaluating each subproblem only once. There may be more than one correct order!

5. Compute the asymptotic complexity of your algorithm. This will be independent of your evaluation order.