Dynamic Programming

Dynamic Programming

- In Dynamic Programming, "Programming" refers to a tabular method, not to writing computer code.

Dynamic Programming

- In Dynamic Programming, "Programming" refers to a tabular method, not to writing computer code.

- Divide-and-conquer algorithms partition the problem into disjoint subproblems (as well as independent), solve the subproblems recursively, and then combine their solutions to solve the original problem.

Dynamic Programming

- In Dynamic Programming, "Programming" refers to a tabular method, not to writing computer code.

- Divide-and-conquer algorithms partition the problem into disjoint subproblems (as well as independent), solve the subproblems recursively, and then combine their solutions to solve the original problem.

- In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.

Dynamic Programming

- In Dynamic Programming, "Programming" refers to a tabular method, not to writing computer code.

- Divide-and-conquer algorithms partition the problem into disjoint subproblems (as well as independent), solve the subproblems recursively, and then combine their solutions to solve the original problem.

- In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems.

- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. E.g.

  Take Fibonacci recurrence: $f(n) = f(n-1) + f(n-2)$

  $f(8) = f(7) + f(6) = (f(6) + f(5)) + f(6)$

Dynamic Programming

- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Dynamic Programming

- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

- DC is a top-down method. When a problem is solved by DC, we immediately attack the complete instance, which we then divide into smaller and smaller subinstances as the algorithm progresses.

Dynamic Programming

- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

- DC is a top-down method. When a problem is solved by DC, we immediately attack the complete instance, which we then divide into smaller and smaller subinstances as the algorithm progresses.

- Dynamic Programming on the other hand is a bottom-up technique. We usually start with the smallest, and hence the simplest, subinstances.

Dynamic Programming

- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

- DC is a top-down method. When a problem is solved by DC, we immediately attack the complete instance, which we then divide into smaller and smaller subinstances as the algorithm progresses.

- Dynamic Programming on the other hand is a bottom-up technique. We usually start with the smallest, and hence the simplest, subinstances.

- By combining their solutions, we obtain the answers to subinstances of increasing size, until finally we arrive at the solution of the original instance.

Dynamic Programming

- We typically apply dynamic programming to optimization problems.

Dynamic Programming

- We typically apply dynamic programming to optimization problems.

- Such problems can have many possible solutions.

Dynamic Programming

- We typically apply dynamic programming to optimization problems.

- Such problems can have many possible solutions.

- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Dynamic Programming

- We typically apply dynamic programming to optimization problems.

- Such problems can have many possible solutions.

- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

- We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

Dynamic Programming

- Matrix-chain Multiplication:

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \cdots A_n \,. \tag{15.5}$$

We can evaluate the expression (15.5) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

Dynamic Programming

- Matrix-chain Multiplication:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

Dynamic Programming

- Matrix-chain Multiplication:

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr. In what follows, we shall express costs in terms of the number of scalar multiplications.

Dynamic Programming

- Matrix-chain Multiplication:

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. If we multiply according to the parenthesization $((A_1 A_2) A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1 (A_2 A_3))$, we perform $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

Dynamic Programming

- Matrix-chain Multiplication:

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Dynamic Programming

- Matrix-chain Multiplication:

For convenience, let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1, n]$.

Dynamic Programming

- Matrix-chain Multiplication:

Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k}A_{k+1..j}$ takes $p_{i-1}p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j \ .$$

This recursive equation assumes that we know the value of k, which we do not. There are only $j-i$ possible values for k, however, namely $k = i, i+1, \ldots, j-1$. Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \ . \end{cases} \qquad (15.7)$$

Dynamic Programming

- Matrix-chain Multiplication:

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Dynamic Programming

- Matrix-chain Multiplication:

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Dynamic Programming

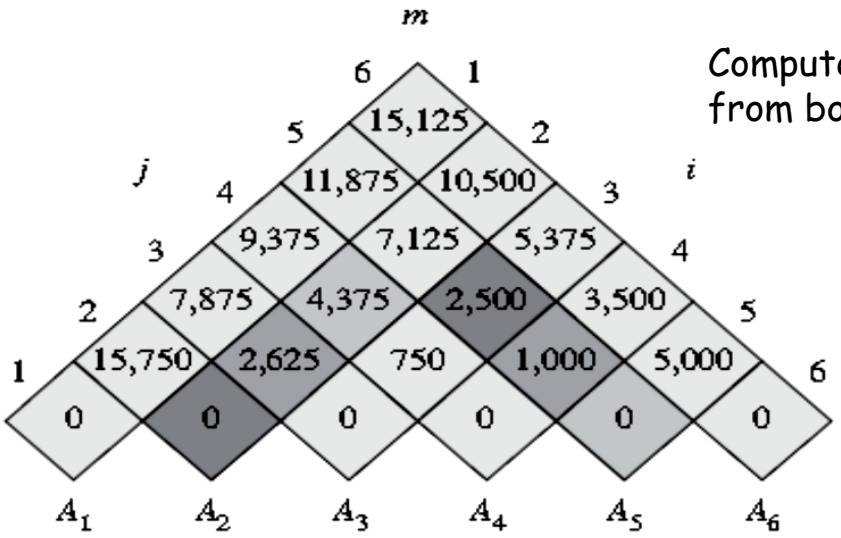
- Matrix-chain Multiplication:

The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute $m[i, i + 1]$ for $i = 1, 2, \ldots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 5–13. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 10–13 depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Dynamic Programming

- Matrix-chain Multiplication: $$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$



Computed row wise from bottom to top

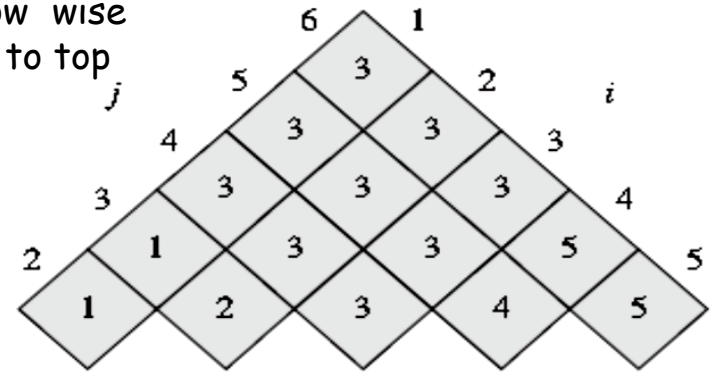


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125.$$

Dynamic Programming

- Matrix-chain Multiplication:

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Dynamic Programming

- 0-1 Knapsack Problem

As in Section 6.5, we are given a number of objects and a knapsack. This time, however, we suppose that the objects may *not* be broken into smaller pieces, so we may decide either to take an object or to leave it behind, but we may not take a fraction of an object. For $i = 1, 2, \ldots, n$, suppose that object i has a positive weight w_i and a positive value v_i. The knapsack can carry a weight not exceeding W. Our aim is again to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint. Let x_i be 0 if we elect not to take object i, or 1 if we include object i. In symbols the new problem may be stated as:

$$\text{maximize} \sum_{i=1}^{n} x_i v_i \qquad \text{subject to} \sum_{i=1}^{n} x_i w_i \leq W$$

where $v_i > 0$, $w_i > 0$ and $x_i \in \{0, 1\}$ for $1 \leq i \leq n$. Here the conditions on v_i and w_i are constraints on the instance; those on x_i are constraints on the solution.

Dynamic Programming

- 0-1 Knapsack Problem

Unfortunately the greedy algorithm turns out not to work when x_i is required to be 0 or 1. For example, suppose we have three objects available, the first of which weighs 6 units and has a value of 8, while the other two weigh 5 units each and have a value of 5 each. If the knapsack can carry 10 units, then the optimal load includes the two lighter objects for a total value of 10. The greedy algorithm, on the other hand, would begin by choosing the object that weighs 6 units, since this is the one with the greatest value per unit weight. However if objects cannot be broken the algorithm will be unable to use the remaining capacity in the knapsack. The load it produces therefore consists of just one object with a value of only 8.

To solve the problem by dynamic programming, we set up a table $V[1..n, 0..W]$, with one row for each available object, and one column for each weight from 0 to W. In the table, $V[i, j]$ will be the maximum value of the objects we can transport if the weight limit is j, $0 \leq j \leq W$, and if we only include objects numbered from 1 to i, $1 \leq i \leq n$. The solution of the instance can therefore be found in $V[n, W]$.

Dynamic Programming

- Longest Common Subsequence
  - A subsequence of a string S, is a set of characters that appear in left to-right order, but not necessarily consecutively.

Dynamic Programming

- Longest Common Subsequence
  - A subsequence of a string S, is a set of characters that appear in left to-right order, but not necessarily consecutively.

  - Example: ACTTGCG
    - ACT, ATTC, T, ACTTGC are all subsequences.
    - TTA is not a subequence

Dynamic Programming

- Longest Common Subsequence
  - A subsequence of a string S, is a set of characters that appear in left to-right order, but not necessarily consecutively.
  - Example: ACTTGCG
    - ACT, ATTC, T, ACTTGC are all subsequences.
    - TTA is not a subequence
  - For another example, Z = {B, C, D, B} is a subsequence of X = {A, B, C, B, D, A, B} with corresponding index sequence {2, 3, 5, 7}.

Dynamic Programming

- Longest Common Subsequence
  - A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.

Dynamic Programming

- Longest Common Subsequence
  - A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.
  - Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y .

Dynamic Programming

- Longest Common Subsequence
  - A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.
  - Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y .
  - For example, if X = {A, B, C, B, D, A, B} and Y = {B, D, C, A, B, A}, the sequence {B, C, A} is a common subsequence of both X and Y . The sequence {B, C, A} is not a longest common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence {B, C, B, A}, which is also common to both X and Y , has length 4.

Dynamic Programming

- Longest Common Subsequence
  - A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.
  - Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y .
  - For example, if X = {A, B, C, B, D, A, B} and Y = {B, D, C, A, B, A}, the sequence {B, C, A} is a common subsequence of both X and Y . The sequence {B, C, A} is not a longest common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence {B, C, B, A}, which is also common to both X and Y , has length 4.
  - The sequence {B, C, B, A} is an LCS of X and Y , as is the sequence {B, D, A, B}, since X and Y have no common subsequence of length 5 or greater.

Dynamic Programming

- Longest Common Subsequence
  - In the longest-common-subsequence problem, we are given two sequences X = {x1, x2, … , xm} and Y = {y1, y2, …, yn} and wish to find a maximum-length common subsequence of X and Y.

  - Let us define c[i, j] to be the length of an LCS of the sequences Xi and Yj .
  - If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0.
  - The optimal substructure of the LCS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Dynamic Programming

- Longest Common Subsequence

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

X = {A, B, C, B, D, A, B} and
Y = {B, D, C, A, B, A}.

Dynamic Programming

- Longest Common Subsequence

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Figure 15.8 shows the tables produced by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Dynamic Programming

- Longest Common Subsequence

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

The procedure takes time O(m + n), since it decrements at least one of i and j in each recursive call.

Dynamic Programming

- Principle of Optimality:
  - It states that - in an optimal sequence of decisions or choices, each subsequence must also be optimal.

- Optimal Substructure Property
  - a problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

- Tabulation vs Memoization

Disclaimer

- Contents of this presentation are not original and they have been prepared from various sources just for the teaching purpose.