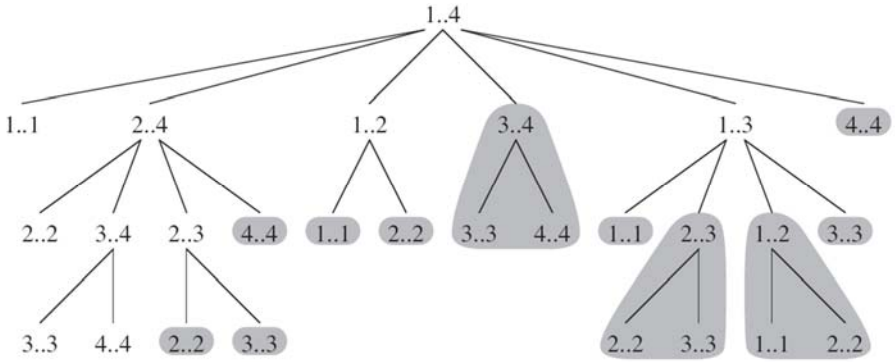
Illustration (1/2)

- The (inefficient) recursive procedure that determines $m[i, j]$:

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```


Illustration (2/2)

- The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4):



The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

Reconstructing an Optimal Solution

- We often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored.

Memoization (1/5)

- There is an alternative approach to dynamic programming that often offers the efficiency of the bottom-up dynamic-programming approach while maintaining a top-down strategy.
- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem.
  - Each table entry initially contains a special value to indicate that the entry has yet to be filled in.
  - When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table.

Memoization (2/5)

- Each subsequent time that we encounter this subproblem, we simply look up the value stored in the table and return it.

- Memoized Version of RECURSIVE-MATRIX-CHAIN:

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```


Memoization (3/5)

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```


Memoization (4/5)

- In summary, we can solve the matrix-chain multiplication problem by either a top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming algorithm in $O(n^3)$ time.
- Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblems are repeatedly solved.

Memoization (5/5)

- In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table.
- If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Outine

- Rod Cutting
- Matrix-Chain Multiplication
- Elements of Dynamic Programming
- ***Longest Common Subsequence***
- Optimal Binary Search Trees

Introduction (1/3)

- Biological applications often need to compare the DNA of two (or more) different organisms.
  - One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.
- we can express a strand of DNA as a string over the finite set {A, C, G, T}.
  - For example, the DNA of one organism may be S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.

Introduction (2/3)

- We can define similarity in many different ways:
  - We can say that two DNA strands are similar if one is a substring of the other. (Chapter 32)
  - We could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 15-5)
  - Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are.

Introduction (3/3)

S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA

What should S_3 be?

➡ S_3 = GTCGTCGGAAGCCGGCCGAA

- We formalize this last notion of similarity as the longest-common-subsequence problem.
  - A subsequence of a given sequence is just the given sequence with zero or more elements left out.