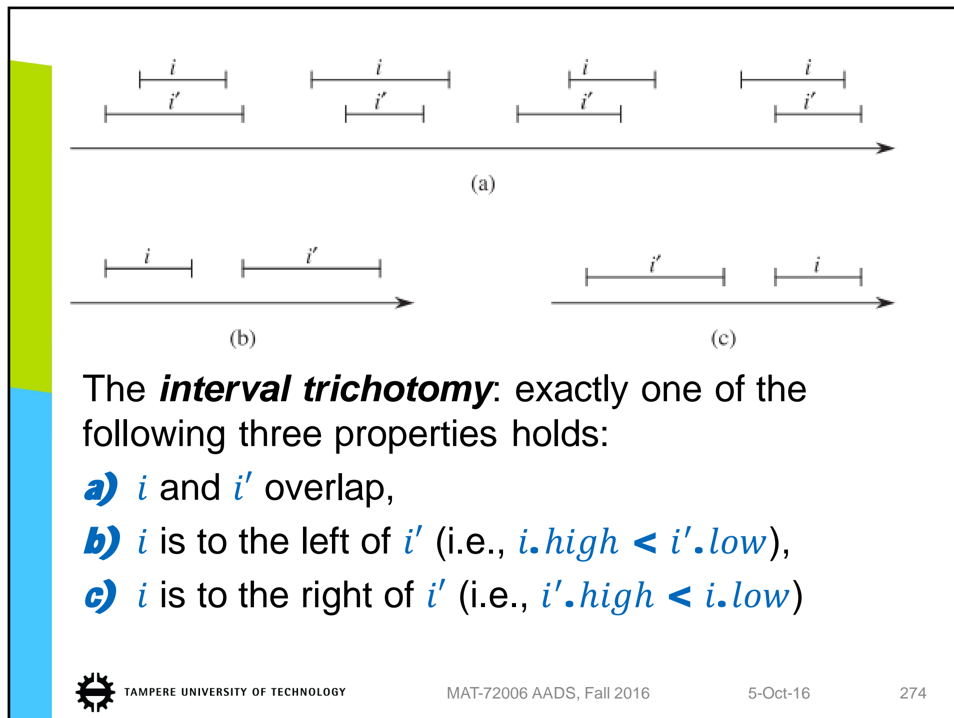
14.3 Interval trees

- We augment RBTs to support operations on dynamic sets of intervals
- A ***closed interval*** is an ordered pair of real numbers $[t_1, t_2]$, with $t_1 \leq t_2$
- Interval $[t_1, t_2]$ represents the set $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$
- ***Open*** and ***half-open*** intervals omit both or one of the endpoints from the set, respectively
- Extending the following results to open and half-open intervals is conceptually straightforward

- Intervals are convenient for representing events that each occupy a continuous period of time
- We might, e.g., wish to query a database of time intervals to find out what events occurred during a given interval
- The data structure to follow provides an efficient means for maintaining such an interval database
- We can represent an interval $[t_1, t_2]$ as an object i, with attributes $i.low = t_1$ (the low endpoint) and $i.high = t_2$ (the high endpoint)
- We say that intervals i and i' overlap if $i \cap i' \neq \emptyset$
  i.e., if $i.low \leq i'.high$ and $i'.low \leq i.high$



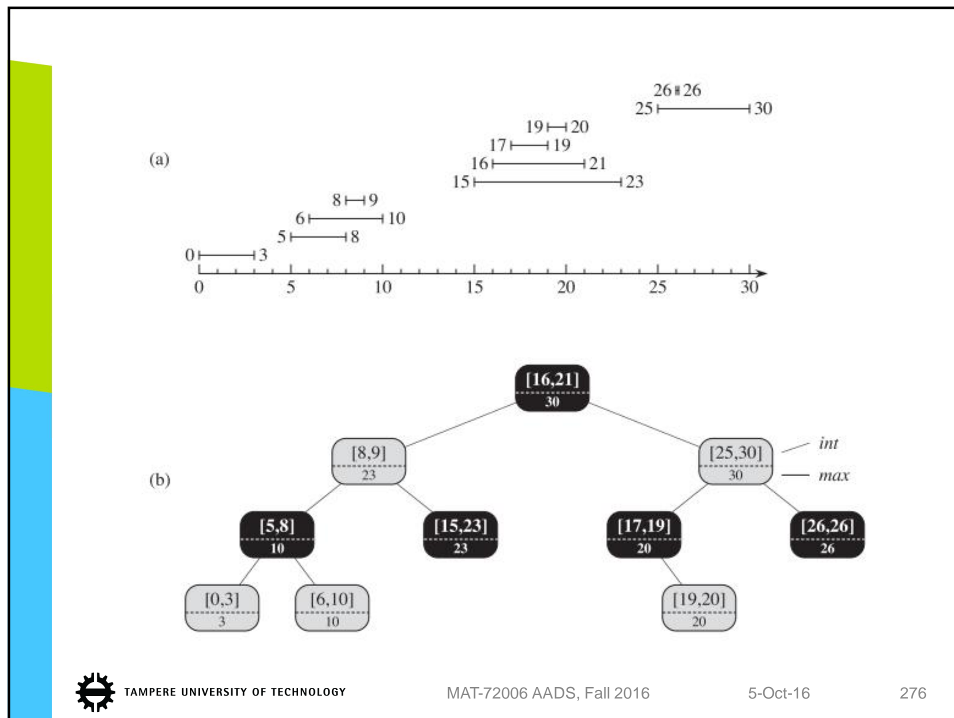
- An interval tree is a RBT that maintains a dynamic set of elements, with each element x containing an interval $x.int$

INTERVAL-INSERT(T, x) adds the element x, whose int attribute is assumed to contain an interval, to the interval tree T

INTERVAL-DELETE (T, x) removes the element x from T

INTERVAL-SEARCH (T, i) returns a pointer to an element x in T such that $x.int$ overlaps interval i, or a pointer to the sentinel $T.\mathbf{nil}$ if no such element is in the set

TAMPERE UNIVERSITY OF TECHNOLOGY MAT-72006 AADS, Fall 2016 5-Oct-16 275



INTERVAL-SEARCH(T, i)

1. $x \leftarrow T.root$
2. **while** $x \neq T.nil$ **and** i does not overlap $x.int$
3. **if** $x.left \neq T.nil$ **and** $x.left.\mathbf{max} \geq i.low$
4. $x \leftarrow x.left$
5. **else** $x \leftarrow x.right$
6. **return** x

- Each iteration of the basic loop takes $O(1)$ time
- The height of an n-node RBT is $O(\lg n)$
- INTERVAL-SEARCH procedure takes $O(\lg n)$ time

- INTERVAL-SEARCH on the previous interval tree:
- We wish to find an interval that overlaps the interval i = **[22,25]**
- We begin with x as the root, which contains **[16,21]** and does not overlap i
- $x.left.\mathbf{max} = \mathbf{23} \geq i.low = \mathbf{22}$ and the loop continues with x as the left child—the node containing **[8,9]**, which does not overlap i
- This time, $x.left.\mathbf{max} = \mathbf{10} \leq i.low = \mathbf{22}$, and so the loop continues with the right child of x
- Because the interval **[15,23]** stored in this node overlaps i, the procedure returns this node

- An unsuccessful search: we wish to find an interval that overlaps i = **[11,14]**
- Since the root's interval **[16,21]** does not overlap i, and since $x.left.\mathbf{max} = \mathbf{23} \geq i.low = \mathbf{11}$, we go left to the node containing **[8,9]**
- Interval **[8,9]** doesn't overlap i, and $x.left.\mathbf{max} = \mathbf{10} \leq i.low = \mathbf{11}$, and so we go right
- No interval in the left subtree overlaps i
- Interval **[15,23]** does not overlap i, and its left child is $T.nil$, so again we go right, the loop terminates, and we return the sentinel $T.nil$

- To see why INTERVAL-SEARCH is correct, we must understand why it suffices to examine a single path from the root
- The basic idea is that at any node x, if $x.int$ does not overlap i, the search always proceeds in a safe direction:
  - The search will definitely find an overlapping interval if the tree contains one

IV Advanced Design and Analysis Techniques

Dynamic Programming
Greedy Algorithms
Amortized Analysis

15 Dynamic Programming

- Divide-and-conquer algorithms partition the problem into disjoint subproblems, recurse, and then combine the solutions
- Dynamic programming applies when the subproblems overlap—that is, when they share subsubproblems
- A dynamic-programming algorithm solves each subsubproblem just once and saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

- We typically apply dynamic programming to ***optimization problems***, which can have many possible solutions
- Each solution has a value, and we wish to find a solution with the optimal (min or max) value
- We call such a solution ***an*** optimal solution, several solutions may achieve the optimal value
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution, typically in a bottom-up fashion
  4. Construct an optimal solution from computed information

15.1 Rod cutting

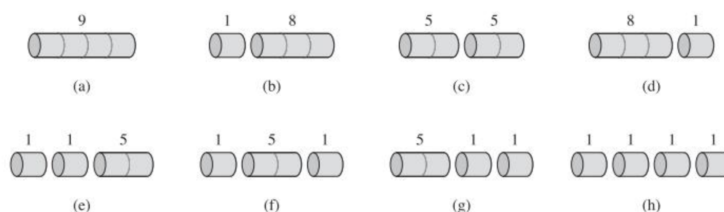
- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells
- Each cut is free
- The management wants to know the best way to cut up the rods
- We assume that we know, for $i = 1,2,\ldots$, the price p_i in dollars that Serling charges for a rod of length i inches
- Rod lengths are always an integral number

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

The **rod-cutting problem** is the following:

- Given a rod of length n inches and a table of prices p_i for $i = 1,2,\ldots,n$,
- determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces

- Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all



- We can cut up a rod of length n in 2^{n-1} different ways: we have an independent option of cutting, or not cutting, at distance i inches from the left end, $i = 1, 2, \ldots, n-1$
- When $n = 4$, there are $2^3 = 8$ ways to cut up the rod, including the way with no cuts at all
- Cutting a 4-inch rod into two 2-inch pieces produces optimal revenue $p_2 + p_2 = 5 + 5 = 10$

- If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

- We can frame the values r_n for $n - 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

- To solve the original problem of size n, we solve smaller problems of the same type
- Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem
- The overall optimal solution incorporates optimal solutions to the related two subproblems, maximizing revenue from each of those pieces
- The rod-cutting problem exhibits ***optimal substructure***: optimal solutions incorporate optimal solutions to related subproblems, which we may solve independently

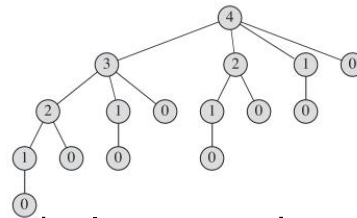
- We view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$
- Only the remainder, and not the first piece, may be further divided
- Decomposition of a length-n rod has a first piece followed by some decomposition of the rest
- No cuts at all: first piece has size $i = n$ and revenue p_n and the remainder has size 0 with corresponding revenue $r_0 = 0$
- We thus obtain the following equation:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

- The following procedure implements the computation implicit in equation in a straightforward, top-down, recursive manner

CUT-ROD(p, n)
1. **if** $n = 0$
2. **return** 0
3. $q \leftarrow -\infty$
4. **for** $i \leftarrow 1$ **to** n
5. $q \leftarrow \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
6. **return** q

- CUT-ROD takes as input array $p[1..n]$ of prices and an integer n
- If we ran CUT-ROD on a computer, we would find that once the input size becomes moderately large, our program would take a long time to run
- For $n = 40$, we would find that our program takes at least several minutes, and most likely more than an hour
- In fact, we would find that each time we increase n by 1, our program's running time would approximately double



- CUT-ROD calls itself recursively over and over again with the same parameter values
  - it solves the same subproblems repeatedly
- CUT-ROD(p, n) calls CUT-ROD$(p, n-i) \equiv$ CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \dots, n-1$
- The amount of work done, as a function of n, grows explosively

- Let $T(n)$ denote the total number of calls made to CUT-ROD when called with its second parameter equal to n
- This equals the number of nodes in a subtree whose root is labeled n in the recursion tree
- The count includes the initial call at its root
- Thus, $T(0) = 1$ and

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

- $T(n) = 2^n$, the running time is exponential

Using dynamic programming for optimal rod cutting

- We arrange for each subproblem to be solved only once, saving its solution
- We can just look the solution up again later
- Dynamic programming serves an example of a ***time-memory trade-off***
- This approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such in polynomial time

- In ***top-down approach with memoization***, we write the procedure recursively modified to save the result of each subproblem
- The procedure now first checks to see whether it has previously solved this subproblem
  - If so, it returns the saved value, if not, the it computes the value in the usual manner

MEMO-CUT-ROD(p, n)

1. let $r[0..n]$ be a new array
2. **for** $i \leftarrow 1$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMO-CUT-ROD-AUX(p, n, r)

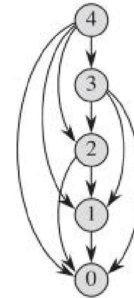
MEMO-CUT-ROD-AUX(p, n, r)

1. **if** $r[n] \geq 0$
2. **return** $r[n]$
3. **if** $n = 0$
4. $q \leftarrow 0$
5. **else** $q \leftarrow -\infty$
6. **for** $i \leftarrow 1$ **to** n
7. $q \leftarrow \max(q, p[i] +$ MEMO-CUT-ROD-AUX$(p, n - i, r))$
8. $r[n] \leftarrow q$
9. **return** q

- The ***bottom-up method*** typically depends on some natural notion of the “size” of a subproblem
- We sort the subproblems by size and solve them in size order, smallest first
- When solving a subproblem, we have already solved all of the smaller subproblems its solution depends upon, we have saved their solutions
- We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems

BOTTOM-UP-CUT-ROD(p, n)

1. let $r[0..n]$ be a new array
2. $r[0] \leftarrow 0$
3. **for** $j \leftarrow 1$ **to** n
4. $\quad q \leftarrow -\infty$
5. $\quad$ **for** $i \leftarrow 1$ **to** j
6. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
7. $\quad r[j] \leftarrow q$
8. **return** $r[n]$



- The running time of BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure
  - The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series
- The running time of its top-down counterpart, MEMO-CUT-ROD, is also $\Theta(n^2)$
  - A recursive call to previously solved subproblem returns immediately, MEMO-CUT-ROD solves each subproblem just once
  - To solve a subproblem of size n, the **for** loop of lines 6–7 iterates n times
  - The total number of iterations of this loop, over all recursive calls forms an arithmetic series

Reconstructing a solution

- The solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes
- We can extend the dynamic-programming approach to record also a choice that led to the optimal value
- An extended version of BOTTOM-UP-CUT-ROD computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

1. let $r[0..n]$ and $s[0..n]$ be new arrays
2. $r[0] \leftarrow 0$
3. **for** $j \leftarrow 1$ **to** n
4. $\quad q \leftarrow -\infty$
5. $\quad$ **for** $i \leftarrow 1$ **to** j
6. $\quad\quad$ **if** $q < p[i] + r[j - i]$
7. $\quad\quad\quad q \leftarrow p[i] + r[j - i]$
8. $\quad\quad\quad s[j] \leftarrow i$
9. $\quad r[j] \leftarrow q$
10. **return** r and s

- The following procedure prints out the complete list of piece sizes in an optimal decomposition of a rod of length n:

PRINT-CUT-ROD-SOLUTION(p, n)

1. $(r, s) \leftarrow$ EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2. **while** $n > 0$
3. print $s[n]$
4. $n \leftarrow n - s[n]$

- In our example, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30
s[i]	0	1	2	3	2	2	6	1	2	3	10

- A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7

15.2 Matrix-chain multiplication

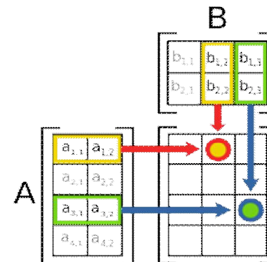
- Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices we wish to compute the product
  $$A_1 A_2 \cdots A_n$$
- We can evaluate the expression using standard algorithm for multiplying pairs of matrices once we have parenthesized it to resolve all ambiguities in how the matrices 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, we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:
  $$(A_1(A_2(A_3 A_4)))$$
  $$(A_1((A_2 A_3)A_4))$$
  $$((A_1 A_2)(A_3 A_4))$$
  $$((A_1(A_2 A_3))A_4)$$
  $$(((A_1 A_2)A_3)A_4)$$

- How we parenthesize a chain of matrices has a dramatic impact on cost of product evaluation
- Standard algorithm for multiplying two matrices:

MATRIX-MULTIPLY(A, B)

1. **if** A.columns $\neq B$.rows
2. **error** "incompatible dimensions"
3. **else** let C be a new A.rows $\times$ B.columns matrix
4. **for** $i \leftarrow 1$ **to** A.rows
5. **for** $j \leftarrow 1$ **to** B.columns
6. $c_{ij} \leftarrow 0$
7. **for** $k \leftarrow 1$ **to** A.columns
8. $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
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
- We shall express costs in terms of the number of scalar multiplications

- Consider matrix product of a chain $\langle A_1, A_2, A_3 \rangle$ of matrices with dimensions **10 × 100**, **100 × 5**, and **5 × 50**
- If we apply the parenthesization $((A_1A_2)A_3)$, we perform **10 · 100 · 5 = 5000** scalar multiplications to compute the **10 × 5** matrix product A_1A_2, plus **10 · 5 · 50 = 2500** further ones to multiply this matrix by A_3

  ∴ a total of **7500** scalar multiplications
- If instead we use $(A_1(A_2A_3))$, we perform **100 · 5 · 50 = 25,000** scalar multiplications to compute the **100 × 50** matrix product A_2A_3, plus another **10 · 100 · 50 = 50,000** scalar multiplications to multiply A_1 by this matrix

  ∴ a total of **75,000** scalar multiplications
- Thus, computing the product according to the first parenthesization is **10** times faster

- ***Matrix-chain multiplication problem***: given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices, where matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications
- We are not actually multiplying matrices
- Our goal is only to determine an order for multiplying matrices that has the lowest cost
- Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications

Counting the number of parentheszations

- Exhaustive checking of all possible parentheses combinations doesn't yield an efficient algorithm
- Let the number of alternative parenthesizations of a sequence of n matrices be $P(n)$
- When $n = 1$, we have just one matrix and only one way to fully parenthesize the matrix product
- When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two may occur between the kth and $(k + 1)$st matrices $\forall k$

- Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

- **Exercise:** show that the solution to the recurrence is $\Omega(2^n)$
- The number of solutions is exponential in n
- The brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain

Applying dynamic programming

Step 1: The structure of an optimal parenthesization

- Let $A_{i..j}$, $i \leq j$ denote the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$
- If $i < j$ then to parenthesize $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some $i \leq k < j$
- I.e., 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: cost of computing the matrix $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of multiplying them together
- The optimal substructure of this problem:
- Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}
- The way we parenthesize $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$
- If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce a way to parenthesize $A_i A_{i+1} \cdots A_j$ with lower cost than the optimum: a contradiction

- Construct an optimal solution to the problem from optimal solutions to subproblems
- Solution to a nontrivial instance requires us to split the product, and any optimal solution contains within it optimal solutions to subproblem instances
- Build an optimal solution by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), finding optimal solutions to these, and then combining these optimal subproblem solutions
- Ensure that in searching for the place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one

Step 2: A recursive solution

- Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems
- For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$
- 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 $m[1, n]$

- If $i = j$, no scalar multiplications are necessary
- Thus, $m[i, i] = 0$ for $i = 1,2, \ldots, n$
- To compute $m[i, j]$, $i < j$, we take advantage of the structure of an optimal solution from step 1
- Assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}
- Then, $m[i, j]$ = the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$ + the cost of multiplying these two matrices together
- 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 $k = i, i + 1, \ldots, j - 1$
- The optimal parenthesization must use one of these values, 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

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

- The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide 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
  $$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

Step 3: Computing the optimal costs

- We could easily write a recursive algorithm based on the recurrence to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \cdots A_n$
- This algorithm takes exponential time
- It is no better than the brute-force method of checking each way of parenthesizing a product
- There are relatively few distinct subproblems: one for each choice of i and j satisfying $1 \leq i \leq j \leq n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all
- We encounter each subproblem many times in different branches of its recursion tree

- We implement the tabular, bottom-up method in MATRIX-CHAIN-ORDER, which assumes that matrix A_i has dimensions $p_{i-1} \times p_i$
- Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$
- The procedure uses
  - table $m[1..n, 1..n]$ for storing the $m[i,j]$ costs and
  - table $s[1..n-1, 2..n]$ records which index of k achieved the optimal cost in computing $m[i,j]$
- We use the table s to construct an optimal solution

- The cost $m[i,j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing matrix-chain products of $\leq j - i + 1$ matrices
- I.e., for $k = i, i + 1, \ldots, j - 1$, $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices
- We fill in m in a manner that corresponds to solving the problem on matrix chains of increasing length
- For $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain

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. $\quad m[i,i] = 0$
5. **for** $l = 2$ **to** n // l is the chain length
6. $\quad$ **for** $i = 1$ **to** $n - l + 1$
7. $\quad\quad j = i + l - 1$
8. $\quad\quad m[i,j] = \infty$
9. $\quad\quad$ **for** $k = i$ **to** $j - 1$
10. $\quad\quad\quad q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
11. $\quad\quad\quad$ **if** $q < m[i,j]$
12. $\quad\quad\quad\quad m[i,j] = q$
13. $\quad\quad\quad\quad s[i,j] = k$
14. **return** m and s

- The algorithm computes in lines 5–13 the minimum costs for chains of length $l = 2$, $m[i, i+1]$ for $i = 1, 2, \dots, n-1$

  during the first execution of the **for** loop
- The second time through the loop, it computes the minimum costs for chains of length $l = 3$, $m[i, i+2]$, 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

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

PRINT-OPTIMAL-PARENS$(s, 1,6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$

- 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
- $m[2,5] =$

$$\min \begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 \\ m[2,3] + m[4,5] + p_1p_3p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1p_4p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$

- The nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm
  - The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n - 1$ values
- The running time of this algorithm is in fact also $\Omega(n^3)$
- The algorithm requires $\Theta(n^2)$ space to store the two tables
- MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one

Step 4: Constructing an optimal solution

- Table $s[1..n-1, 2..n]$ gives us the information we need to multiply the matrices
- Entry $s[i,j]$ records a value of k s.t. an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}
- Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$
- $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1,n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$

- The following procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j
- The call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$

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 ")"