13.2 Rotations

**LEFT-ROTATE**(\(T, x\))

1. \(y = x.right\)  // set \(y\)
2. \(x.right = y.left\) // \(y\)’s left subtree → \(x\)’s right subtree
3. if \(y.left \neq T.nil\)
4. \(y.left.p = x\)
5. \(y.p = x.p\)  // link \(x\)’s parent to \(y\)
6. if \(x.p == T.nil\)
7. \(T.root = y\)
8. elseif \(x == x.p.left\)
9. \(x.p.left = y\)
10. else \(x.p.right = y\)
11. \(y.left = x\)  // put \(x\) on \(y\)’s left
12. \(x.p = y\)
### 13.3 Insertion

**RB-INSERT**(*T, z*)

1. \( y = T.\text{nil} \)
2. \( x = T.\text{root} \)
3. while \( x \neq T.\text{nil} \)
4. \( y = x \)
5. if \( z.\text{key} < x.\text{key} \)
6. \( x = x.\text{left} \)
7. else \( x = x.\text{right} \)
8. \( z.\text{p} = y \)
9. if \( y == T.\text{nil} \)
10. \( T.\text{root} = z \)
11. elseif \( z.\text{key} < y.\text{key} \)
12. \( y.\text{left} = z \)
13. else \( y.\text{right} = z \)
14. \( z.\text{left} = T.\text{nil} \)
15. \( z.\text{right} = T.\text{nil} \)
16. \( z.\text{color} = \text{RED} \)
17. \( \text{RB-INSERT-FIXUP}(T,z) \)

1. All instances of **NIL** in **TREE-INSERT** are replaced by \( T.\text{nil} \)
2. We set \( z.\text{left} \) and \( z.\text{right} \) to \( T.\text{nil} \) in lines 14–15 of **RB-INSERT**, in order to maintain the proper tree structure
3. We color \( z \) red in line 16
4. Coloring \( z \) red may cause a violation of one of the red-black properties, hence, we call \( \text{RB-INSERT-FIXUP}(T,z) \) in line 17 to restore the red-black properties by recoloring nodes and performing rotations
Which of the red-black properties might be violated upon the call to RB-INSERT-FIXUP?

- Property 1 certainly continues to hold, as does property 3, since both children of the newly inserted red node are the sentinel $T.nil$
- Property 5 (equal number of black nodes on every path) is satisfied as well, because node $z$ replaces the (black) sentinel, and node $z$ is red with sentinel children
- The only properties that might be violated are property 2 (the root is black) and property 4 (a red node cannot have a red child)
- Both possible violations are due to $z$ being colored red
- Property 2 is violated if $z$ is the root, and property 4 is violated if $z$’s parent is red

A node $z$ after insertion
- Both $z$ and $z.p$ are red, a violation of property 4 occurs
- Uncle $y$ is red

We recolor nodes and move the pointer $z$ up the tree, resulting in the tree shown
• $z$ and its parent are both red, but $z$’s uncle $y$ is black
• $z$ is the right child of $z'$. $p$
• We perform a left rotation

• Now, $z$ is the left child of its parent
• Recoloring and right rotation yield a legal red-black tree
\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]

\[
\]
### Analysis

- The height of a RBT on $n$ nodes is $O(\lg n)$ => lines 1–16 of RB-INSERT take $O(\lg n)$ time
- In -FIXUP, the **while** loop repeats only in case 1, then the pointer $z$ moves two levels up
  - The total number of times the **while** loop can be executed is $O(\lg n)$
- Thus, RB-INSERT takes a total of $O(\lg n)$ time
- It never performs more than two rotations, the **while** loop terminates if case 2 or 3 is executed

### 13.4 Deletion

- First, we need to customize TRANSPLANT that TREE-DELETE calls so that it applies to a RBT:

  **RB-TRANSPLANT**$(T, u, v)$

  1. if $u.p == T.nil$
  2. $T.root = v$
  3. elseif $u == u.p.left$
  4. $u.p.left = v$
  5. else $u.p.right = v$
  6. $v.p = u.p$
The main differences between RB-DELETE and TREE-DELETE:

- Line 1 sets \( y \) to point to node \( z \) when it has < 2 children and is therefore removed. When \( z \) has two children, line 9 sets \( y \) to point to \( z \)'s successor and \( y \) will move into \( z \)'s position in the tree.

- Node \( y \)'s color might change, a variable stores \( y \)'s color before any changes occur. When \( z \) has two children, then \( y \neq z \) and node \( y \) moves into node \( z \)'s original position; line 20 gives \( y \) the same color as \( z \). If \( y \)'s original color was black, then removing or moving \( y \) could cause violations of the red-black properties.
– We keep track of the node $x$ that moves into $y$’s original position. The assignments (lines 4, 7, 11) set $x$ to point to either $y$’s only child or, if $y$ has no children, the sentinel $T.nil$.

– Since node $x$ moves into $y$’s original position, the attribute $x.p$ is always set to point to the original position in the tree of $y$’s parent. Unless $z$ is $y$’s original parent, the assignment to $x.p$ takes place (line 6 of RB-TRANSPLANT). When $y$’s original parent is $z$, we do not want $x.p$ to point to $y$’s parent, we are removing it from the tree. Node $y$ will move up to take $z$’s place, setting $x.p$ to $y$ (line 13) causes $x.p$ to point to the original position of $y$’s parent, even if $x = T.nil$.

– If node $y$ was black, we might have introduced violations of the red-black properties. RB-DELETE-FIXUP restores the red-black properties. If $y$ was red, the red-black properties still hold when $y$ is removed or moved, for the following reasons:

1. No black-heights in the tree have changed.
2. Because $y$ takes $z$’s place in the tree, along with $z$’s color, we cannot have two adjacent red nodes at $y$’s new position. In addition, if $y$ was not $z$’s right child, then $y$’s original right child $x$ replaces $y$. If $y$ is red, then $x$ must be black, and so replacing $y$ by $x$ cannot cause two red nodes to become adjacent.
3. Since $y$ could not have been the root if it was red, the root remains black.
RB-DELETE-FIXUP($T, x$)

1. \textbf{while} $x \neq T.root$ and $x.color = \text{BLACK}$
2. \textbf{if} $x == x.p.left$
3. \textbf{w} = x.p.right

\hspace{1cm} //----------------------------- case 1
4. \textbf{if} $w.color == \text{RED}$
5. \hspace{1cm} $w.color = \text{BLACK}$
6. \hspace{1cm} $x.p.color == \text{RED}$
7. \hspace{1cm} LEFT-ROTATE($T, x.p$)
8. \hspace{1cm} $w = x.p.right$

\hspace{1cm} //----------------------------- case 2
9. \textbf{if} $w.left.color == \text{BLACK}$
10. \hspace{1cm} \textbf{and} $w.right.color == \text{BLACK}$
11. \hspace{1cm} $w.color = \text{RED}$

\hspace{1cm} //----------------------------- case 3
12. \textbf{else} $w.right.color == \text{BLACK}$
13. \hspace{1cm} $w.left.color = \text{BLACK}$
14. \hspace{1cm} $w.color == \text{RED}$
15. \hspace{1cm} RIGHT-ROTATE($T, w$)
16. \hspace{1cm} $w = x.p.right$

\hspace{1cm} //----------------------------- case 4
17. \textbf{else} $w.right.color == \text{BLACK}$
18. \hspace{1cm} $x = T.root$
19. \hspace{1cm} $x.p.color = \text{BLACK}$
20. \hspace{1cm} RIGHT-ROTATE($T, x.p$)
21. \hspace{1cm} $w.right.color = \text{BLACK}$
22. \hspace{1cm} $x = T.root$
23. \hspace{1cm} $x.color = \text{BLACK}$
24. \else (same as then clause with right/left exchanged)
25. \hspace{1cm} $x.color = \text{BLACK}$

\hspace{1cm} //----------------------------- case 5
26. \textbf{else} (same as then clause with right/left exchanged)
27. \hspace{1cm} $x = T.root$
28. \hspace{1cm} $x.color = \text{BLACK}$
29. \hspace{1cm} $w.color = \text{RED}$

\hspace{1cm} //----------------------------- case 6
30. \textbf{else} (same as then clause with right/left exchanged)
31. \hspace{1cm} $x = T.root$
32. \hspace{1cm} $x.color = \text{BLACK}$
33. \hspace{1cm} $w.color = \text{RED}$
34. \hspace{1cm} LEFT-ROTATE($T, x.p$)
35. \hspace{1cm} $x = T.root$
36. \hspace{1cm} $x.color = \text{BLACK}$

\hspace{1cm} //----------------------------- case 7
37. \textbf{else} (same as then clause with right/left exchanged)
38. \hspace{1cm} $x = T.root$
39. \hspace{1cm} $x.color = \text{BLACK}$
40. \hspace{1cm} $w.color = \text{RED}$
41. \hspace{1cm} LEFT-ROTATE($T, x.p$)
42. \hspace{1cm} $x = T.root$
43. \hspace{1cm} $x.color = \text{BLACK}$

• RB-DELETE-FIXUP restores properties 1, 2, and 4
• The goal of the \textbf{while} loop (lines 1–22) is to move the extra black up the tree until
  1. $x$ points to a red-and-black node, in which case we color $x$ (singly) black (line 23);
  2. $x$ points to the root, in which case we simply “remove” the extra black; or
  3. having performed suitable rotations and recolorings, we exit the loop
**Analysis**

- Height of a RBT of \( n \) nodes is \( O(\lg n) \), total cost without the call to -FIXUP is \( O(\lg n) \) time.
- Within -FIXUP, cases 1, 3, and 4 lead to termination after a constant number of color changes and at most three rotations.
- Only in case 2 the `while` loop can be repeated, and then the pointer \( x \) moves up the tree at most \( O(\lg n) \) times, without rotations.
- Thus, -FIXUP takes \( O(\lg n) \) time and performs \( \leq 3 \) rotations, the overall time for RB-DELETE is \( O(\lg n) \).

---

**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

<table>
<thead>
<tr>
<th>Length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

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 \( 1 \leq k \leq n \), 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:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$
• The following procedure implements the computation implicit in equation in a straightforward, top-down, recursive manner

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

• \( \text{CUT-ROD} \) takes as input array \( p[1..n] \) of prices and an integer \( n \)
• If we ran \( \text{CUT-ROD} \) 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 \(j = 0, 1, \ldots, n - 1\)

• 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 \(i = 1\) to \(n\)
3. \(r[i] = -\infty\)
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. \(q = 0\)
5. else \(q = -\infty\)
6. for \(i = 1\) to \(n\)
7. \(q = \max(q, p[i] + MEMO-CUT-ROD-AUX(p, n - i, r))\)

1. \(r[n] = q\)
2. 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. \(r[0] = 0\)
3. for \(j = 1\) to \(n\)
4. \(q = -\infty\)
5. for \(i = 1\) to \(j\)
6. \(q = \max(q, p[i] + r[j - i])\)
7. \(r[j] = q\)
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. $r[0] = 0$
3. for $j = 1$ to $n$
4. $q = -\infty$
5. for $i = 1$ to $j$
6. if $q < p[i] + r[j - i]$
7. $q = p[i] + r[j - i]$
8. $s[j] = i$
9. $r[j] = q$
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 \( n \):

\[
\text{PRINT-CUT-ROD-SOLUTION}(p, n)
\]

1. \((r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)\)

2. while \( n > 0 \)

3. print \( s[n] \)

4. \( n = n - s[n] \)

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

\[
\begin{array}{cccccccccccc}
\hline
i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
r[i] & 0 & 1 & 5 & 8 & 9 & 10 & 17 & 17 & 20 & 24 & 30 \\
s[i] & 0 & 1 & 2 & 3 & 2 & 2 & 6 & 1 & 2 & 3 & 10 \\
\hline
\end{array}
\]

• A call to \( \text{PRINT-CUT-ROD-SOLUTION}(p, 10) \) 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) \( A_1, A_2, \ldots, A_n \) 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:

\[
\text{MATRIX-MULTIPLY}(A, B)
\]

1. if \( A.\text{columns} \neq B.\text{rows} \) then
2. \hspace{1em} error “incompatible dimensions”
3. else let \( C \) be a new \( A.\text{rows} \times B.\text{columns} \) matrix
4. \hspace{1em} for \( i = 1 \) to \( A.\text{rows} \)
5. \hspace{2em} for \( j = 1 \) to \( B.\text{columns} \)
6. \hspace{3em} \( c_{ij} = 0 \)
7. \hspace{2em} for \( k = 1 \) to \( A.\text{columns} \)
8. \hspace{3em} \( c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \)
9. \hspace{1em} 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 \( (A_1, A_2, A_3) \) of matrices with dimensions \( 10 \times 100, 100 \times 5, \) and \( 5 \times 50 \)

• If we apply the parenthesization \( ((A_1 A_2) A_3) \), we perform \( 10 \cdot 100 \cdot 5 = 5000 \) scalar multiplications to compute the \( 10 \times 5 \) matrix product \( A_1 A_2 \), plus \( 10 \cdot 5 \cdot 50 = 2500 \) further ones to multiply this matrix by \( A_3 \)
  \( \therefore \) a total of \( 7500 \) scalar multiplications

• If instead we use \( (A_1 (A_2 A_3)) \), we perform \( 100 \cdot 5 \cdot 50 = 25,000 \) scalar multiplications to compute the \( 100 \times 5 \) matrix product \( A_2 A_3 \), plus another \( 10 \cdot 100 \cdot 50 = 50,000 \) scalar multiplications to multiply \( A_1 \) by this matrix
  \( \therefore \) a total of \( 75,000 \) scalar multiplications

• Thus, computing the product according to the first parenthesization is \( 10 \) times faster