Exercises 7.1

6. ▶ The **ancestry problem** asks to determine whether a vertex $u$ is an ancestor of vertex $v$ in a given binary (or, more generally, rooted ordered) tree of $n$ vertices. Design a $O(n)$ input enhancement algorithm that provides sufficient information to solve this problem for any pair of the tree’s vertices in constant time.

**Solution:**

6. Vertex $u$ is an ancestor of vertex $v$ in a rooted ordered tree $T$ if and only if the following two inequalities hold

$$\text{preorder}(u) \leq \text{preorder}(v) \quad \text{and} \quad \text{postorder}(u) \geq \text{postorder}(v),$$

where $\text{preorder}$ and $\text{postorder}$ are the numbers assigned to the vertices by the preorder and postorder traversals of $T$, respectively. Indeed, preorder traversal visits recursively the root and then the subtrees numbered from left to right. Therefore,

$$\text{preorder}(u) \leq \text{preorder}(v)$$

if and only if either $u$ is an ancestor of $v$ (i.e., $u$ is on the simple path from the root’s tree to $v$) or $u$ is to the left of $v$ (i.e., $u$ and $v$ are not on the same simple path from the root to a leaf and $T(u)$ is to the left of $T(v)$ where $T(u)$ and $T(v)$ are the subtrees of the nearest common ancestor of $u$ and $v$, respectively). Similarly, postorder traversal visits recursively the subtrees numbered from left to right and then the root. Therefore,

$$\text{postorder}(u) \geq \text{postorder}(v)$$

if and only if either $u$ is an ancestor of $v$ or $v$ is to the left of $u$. Hence,

$$\text{preorder}(u) \leq \text{preorder}(v) \quad \text{and} \quad \text{postorder}(u) \geq \text{postorder}(v)$$

is necessary and sufficient for $u$ to be an ancestor of $v$.

The time efficiencies of both traversals are in $O(n)$ (Section 4.4); once the preorder and postorder numbers are precomputed, checking the two inequalities takes constant time for any given pair of the vertices.

Exercises 7.2

2. Consider the problem of searching for genes in DNA sequences using Horspool’s algorithm. A DNA sequence consists of a text on the alphabet $\{A, C, G, T\}$ and the gene or gene segment is the pattern.

   a. Construct the shift table for the following gene segment of your chromosome 10:

      TTATAGATCTC

   b. Apply Horspool’s algorithm to locate the above pattern in the following DNA sequence:

      TTATAGATCTC

**Solution:**
a. For the pattern TCCTATTCTT and the alphabet \{A, C, G, T\}, the shift table looks as follows:

<table>
<thead>
<tr>
<th>c</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>t(c)</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

b. Below the text and the pattern, we list the characters of the text that are aligned with the last T of the pattern, along with the corresponding number of character comparisons (both successful and unsuccessful) and the shift size:

the text: TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT
the pattern: TCCTATTCTT

| T: | 2 comparisons, shift 1 |
| C: | 1 comparison, shift 2 |
| T: | 2 comparisons, shift 1 |
| A: | 1 comparison, shift 5 |
| T: | 8 comparisons, shift 1 |
| T: | 3 comparisons, shift 1 |
| T: | 3 comparisons, shift 1 |
| A: | 1 comparison, shift 5 |
| T: | 2 comparisons, shift 1 |
| C: | 1 comparison, shift 2 |
| C: | 1 comparison, shift 2 |
| T: | 2 comparisons, shift 1 |
| A: | 1 comparison, shift 5 |
| T: | 10 comparisons to stop the successful search |

Exercises 7.3
1. For the input 30, 20, 56, 75, 31, 19 and hash function \( h(K) = K \mod 11 \)
   a. construct the open hash table.
   b. find the largest number of key comparisons in a successful search in this table.
   c. find the average number of key comparisons in a successful search in this table.

Solution:

   a. The list of keys: 30, 20, 56, 75, 31, 19
   The hash function: \( h(K) = K \mod 11 \)

   The hash addresses:

   \[
   \begin{array}{cccccc}
   K & 30 & 20 & 56 & 75 & 31 & 19 \\
   h(K) & 8 & 9 & 1 & 9 & 9 & 8 \\
   \end{array}
   \]

   The open hash table:
<table>
<thead>
<tr>
<th></th>
<th>0</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>56</td>
<td></td>
<td>30</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td>19</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

b. The largest number of key comparisons in a successful search in this table is 3 (in searching for \( K = 31 \)).

c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

\[
\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 2 = \frac{10}{6} \approx 1.7.
\]

2. For the input 30, 20, 56, 75, 31, 19 and hash function \( h(K) = K \mod 11 \)
a. construct the open hash table.
b. find the largest number of key comparisons in a successful search in this table.
c. find the average number of key comparisons in a successful search in this table.

Solution:

a. The list of keys: 30, 20, 56, 75, 31, 19
The hash function: \( h(K) = K \mod 11 \)

<table>
<thead>
<tr>
<th>( K )</th>
<th>30</th>
<th>20</th>
<th>56</th>
<th>75</th>
<th>31</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h(K) )</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>75</td>
</tr>
<tr>
<td>31</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>20</td>
<td>75</td>
</tr>
<tr>
<td>31</td>
<td>56</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>20</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

b. The largest number of key comparisons in a successful search is 6 (when searching for \( K = 19 \)).

c. The average number of key comparisons in a successful search in this table, assuming that a search for each of the six keys is equally likely, is

\[
\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 6 = \frac{14}{6} \approx 2.3.
\]

Exercises 8.1
7. Which of the following algorithms for computing a binomial coefficient is most efficient?
   a. Use the formula
      \[ C(n, k) = \frac{n!}{k!(n-k)!}. \]
   b. Use the formula
      \[ C(n, k) = \frac{n(n-1)...(n-k+1)}{k!}. \]
   c. Apply recursively the formula
      \[
      C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{for } n > k > 0, \\
      C(n, 0) = C(n, n) = 1.
      \]
   d. Apply the dynamic programming algorithm.

Solution:
7. a. Assuming that 1! is computed as 0! \cdot 1 and, hence, requires one multiplication, the use of the formula
      \[ C(n, k) = \frac{n!}{k!(n-k)!}. \]
      requires \( n + k + n - k = 2n \) multiplications and one division.
   b. Use the formula
      \[ C(n, k) = \frac{n(n-1)...(n-k+1)}{k!}. \]
      requires only \( k + k = 2k \) multiplications and one division. (We assumed that the product in the numerator is initialized to 1.) Hence, (b) is more efficient than (a). Also note that the overflow problem is less severe for formula (b) than for formula (a). It can be further alleviated by using the formula
      \[
      \frac{n(n-1)...(n-k+1)}{k!} = \frac{n(n-1)...(n-(k-1))}{k!} \cdot \frac{n-(k-1)}{1}.
      \]
   c. Apply recursively the formula
      \[
      C(n, k) = C(n - 1, k - 1) + C(n - 1, k) \quad \text{for } n > k > 0, \\
      C(n, 0) = C(n, n) = 1.
      \]
      requires \( C(n, k) - 1 \) additions (see Problem 6), which is much worse than the dynamic programming algorithm that requires \( nk - \frac{1}{2}k^2 - \frac{1}{2}k \) additions (see Section 8.1).
   d. The efficiency of the dynamic programming algorithm, as explained in the section, is in \( \Theta(nk) \).

Thus, the choice comes to be between algorithms (b) and (d). Making a choice between two algorithms with different basic operations should be conditioned on their relative speeds. Still, since the time efficiency of algorithm (b) is in \( \Theta(k) \) while the time efficiency of algorithm (d) is in \( \Theta(nk) \), we have to conclude that the former is more time efficient, at least
asymptotically, than the latter. Algorithm (b) is obviously more space efficient than algorithm (d), too.

**Exercises 8.2**

6. a. Explain how Warshall’s algorithm can be used to determine whether a given digraph is a dag (directed acyclic graph). Is it a good algorithm for this problem?

   b. Is it a good idea to apply Warshall’s algorithm to find the transitive closure of an undirected graph?

**Solution:**

a. With the book’s definition of the transitive closure (which considers only nontrivial paths of a digraph), a digraph has a directed cycle if and only if its transitive closure has a 1 on its main diagonal. The algorithm that finds the transitive closure by applying Warshall’s algorithm and then checks the elements of its main diagonal is cubic. This is inferior to the quadratic algorithms for checking whether a digraph represented by its adjacency matrix is a dag, which were discussed in Section 5.3.

b. No. If $T$ is the transitive closure of an undirected graph, $T[i,j] = 1$ if and only if there is a nontrivial path from the $i$th vertex to the $j$th vertex. If $i \neq j$, this is the case if and only if the $i$th vertex and the $j$th vertex belong to the same connected component of the graph. Thus, one can find the elements outside the main diagonal of the transitive closure that are equal to 1 by a depth-first search or a breadth-first search traversal, which is faster than applying Warshall’s algorithm. If $i = j$, $T[i,i] = 1$ if and only if the $i$th vertex is not isolated, i.e., if it has an edge to at least one other vertex of the graph. Isolated vertices, if any, can be easily identified by the graph’s traversal as one-node connected components of the graph.

7. Solve the all-pairs shortest path problem for the digraph with the following weight matrix

\[
\begin{bmatrix}
0 & 2 & \infty & 1 & 8 \\
6 & 0 & 3 & 2 & \infty \\
\infty & \infty & 0 & 4 & \infty \\
\infty & \infty & 2 & 0 & 3 \\
3 & \infty & \infty & \infty & 0
\end{bmatrix}
\]

**Solution:**

Applying Floyd’s algorithm to the given weight matrix generates the following sequence of matrices:
Exercises 8.3

10. **Matrix chain multiplication**  Consider the problem of minimizing the total number of multiplications made in computing the product of $n$ matrices

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

whose dimensions are $d_0$ by $d_1$, $d_1$ by $d_2$, ..., $d_{n-1}$ by $d_n$, respectively. (Assume that all intermediate products of two matrices are computed by the brute-force (definition-based) algorithm.

a. Give an example of three matrices for which the number of multiplications in $(A_1 \cdot A_2) \cdot A_3$ and $A_1 \cdot (A_2 \cdot A_3)$ differ at least by a factor 1000.

b. How many different ways are there to compute the chained product of $n$ matrices?

c. Design a dynamic programming algorithm for finding an optimal order of multiplying $n$ matrices.

Solution:

a. Multiplying two matrices of dimensions $\alpha$-by-$\beta$ and $\beta$-by-$\gamma$ by the definition-based algorithm requires $\alpha \beta \gamma$ multiplications. (There are $\alpha \beta \gamma$ elements in the product, each requiring $\beta$ multiplications to be computed.) If the dimensions of $A_1$, $A_2$, and $A_3$ are $d_0$-by-$d_1$, $d_1$-by-$d_2$, and $d_2$-by-$d_3$, respectively, then $(A_1 \cdot A_2) \cdot A_3$ will require

$$d_0 d_1 d_2 + d_0 d_2 d_3 = d_0 d_2 (d_1 + d_3)$$

multiplications, while $A_1 \cdot (A_2 \cdot A_3)$ will need

$$d_1 d_2 d_3 + d_0 d_1 d_3 = d_1 d_3 (d_0 + d_2)$$

multiplications. Here is a simple choice of specific values to make, say, the first of them be 1,000 times larger than the second:
\[ d_0 = d_2 = 10^3, \quad d_1 = d_3 = 1. \]

b. Let \( m(n) \) be the number of different ways to compute a chain product of \( n \) matrices \( A_1 \cdots A_n \). Any parenthesization of the chain will lead to multiplying, as the last operation, some product of the first \( k \) matrices \( (A_1 \cdots A_k) \) and the last \( n-k \) matrices \( (A_{k+1} \cdots A_n) \). There are \( m(k) \) ways to do the former, and there are \( m(n-k) \) ways to do the latter. Hence, we have the following recurrence for the total number of ways to parenthesize the matrix chain of \( n \) matrices:

\[ m(n) = \sum_{k=1}^{n} m(k)m(n-k) \text{ for } n > 1, \quad m(1) = 1. \]

Since parenthesizing a chain of \( n \) matrices for multiplication is very similar to constructing a binary tree of \( n \) nodes, it should come as no surprise that the above recurrence is very similar to the recurrence

\[ b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k) \text{ for } n > 1, \quad b(0) = 1, \]

for the number of binary trees mentioned in Section 8.3. Nor is it surprising that their solutions are very similar, too: namely,

\[ m(n) = b(n - 1) \text{ for } n \geq 1, \]

where \( b(n) \) is the number of binary trees with \( n \) nodes. Let us prove this assertion by mathematical induction. The basis checks immediately: \( m(1) = b(0) = 1 \). For the general case, let us assume that \( m(k) = b(k - 1) \) for all positive integers not exceeding some positive integer \( n \) (we’re using the strong version of mathematical induction); we’ll show that the equality holds for \( n+1 \) as well. Indeed,

\[
m(n+1) = \sum_{k=1}^{n} m(k)m(n+1-k)
\]

\[= \left[\text{using the induction’s assumption}\right]\sum_{k=1}^{n} b(k-1)b(n-k)
\]

\[= \left[\text{substituting } l = k - 1\right]\sum_{l=0}^{n-1} b(l)b(n - 1 - l)
\]

\[= \left[\text{see the recurrence for } b(n)\right] b(n).
\]

c. Let \( M[i,j] \) be the optimal (smallest) number of multiplications needed for computing \( A_i \cdots A_j \). If \( k \) is an index of the last matrix in the first factor of the last matrix product, then

\[
M[i,j] = \max_{1 \leq k \leq j-1} \{M[i,k] + M[k+1,j] + d_i d_k d_j\} \text{ for } 1 \leq i < j \leq n,
\]

\[M[i,i] = 0.
\]

This recurrence, which is quite similar to the one for the optimal binary
search tree problem, suggests filling the $n + 1$-by-$n + 1$ table diagonal by
diagonal as in the following algorithm:

**Algorithm** MatrixChainMultiplication($D[0..n]$)

//Solves matrix chain multiplication problem by dynamic programming
//Input: An array $D[0..n]$ of dimensions of $n$ matrices
//Output: The minimum number of multiplications needed to multiply
//a chain of $n$ matrices of the given dimensions and table $T[1..n, 1..n]$
//for obtaining an optimal order of the multiplications

for $i ← 1$ to $n$ do $M[i, i] ← 0$
for $d ← 1$ to $n - 1$ do  //diagonal count
  for $i ← 1$ to $n - d$ do
    $j ← i + d$
    $minval ← \infty$
    for $k ← i$ to $j - 1$ do
      $temp ← M[i, k] + M[k + 1, j] + D[i - 1] \ast D[k] \ast D[j]$
      if $temp < minval$
        $minval ← temp$
        $kmin ← k$
    $T[i, j] ← kmin$

return $M[1, n], T$

To find an optimal order to multiply the matrix chain, call OptimalMultiplicationOrder($1, n$) below:

**Algorithm** OptimalMultiplicationOrder($i, j$)

//Input: Indices $i$ and $j$ of the first and last matrices in $A_1...A_j$ and
//Output: $A_1...A_j$ parenthesized for optimal multiplication

if $i = j$
  print(“$A_i$”)
else
  $k ← T[i, j]$
  print(“(“)
  OptimalMultiplicationOrder($i, k$)
  OptimalMultiplicationOrder($k + 1, j$)
  print(“)”)