Problems 1–4 and problems 11.57 from the text.
For problems 1–4 give a recursive formulation of the value being minimized or maximized
and explain what is the runtime of the resulting dynamic programming implementation.

Honors Section. Problem 5 below in addition.

1. Recall the Goodness\( (v) \) function defined on Page 627, GN\((v) \) for short.

\[
GN(v) = v.dat + \sum_{w \text{ a child of } v} 2GN(w).
\]

You are given an array Data\([1 : n]\) of integer values, which may include negative values.

a. Suppose the integer values are allocated to the .dat fields of the nodes in an \( n \)-node
binary tree \( T \) in postorder. Your task is to determine the tree \( T \) maximizing GN\((T)\). Give
a recursive formulation of the function PostBinBest\((i, k)\), BPP\((i, k)\) for short, which equals
the largest Goodness value achievable by a \((k - i + 1)\)-node binary tree for the data values
Data\([i : k]\). Now explain why a dynamic programming implementation of this function runs
in \( O(n^3) \) time.

An example is shown below.
Suppose that Data\([1 : 4]\) = \([-2, 4, 1, 3]\). Below we show one possible choice of \( T \):

\[
\begin{array}{c|c}
\text{.dat values} & \text{GN values} \\
\hline
3 & 17 \\
/ \ \ \ \ / \ \\
/ \ \ \ / \ \\
-2 & 1 & -2 & 9 \\
\ \ \ \ \ \ \ \ \ \ \ \\
4 & 4 \\
\end{array}
\]

b. Repeat part a but for general trees, again in postorder. In other words, give a recursive
formulation of the function PostGenBest\((n)\) which equals the largest Goodness value achievable
by an \( n \)-node general tree. Again, explain why a dynamic programming implementation
of this function runs in \( O(n^3) \) time.

2. Consider the problem of finding the optimal weighted binary search tree for \( n \) items
\( e_1 < e_2 < \cdots < e_n \). Each item \( e_i \) has a non-negative integer weight \( w_i \). Suppose item \( e_i \)
is accessed \( r_i \) times. The cost of an access to item \( e_i \) is defined to be the path length \( l_i \),
measured in nodes, in the binary search tree going from the root to the node storing \( e_i \)
times \( w_i \). Your task is to find a binary search tree \( T \) minimizing the overall access cost,
\[
\sum_{i=1}^{n} l_i \cdot w_i \cdot r_i.
\]
Let Cost\((i, k)\) be the cost of the optimal binary search tree storing items
$e_i, e_{i+1}, \ldots, e_k$. Give a recursive formulation for $\text{Cost}(i, k)$. Then explain how to use it to obtain the desired optimal binary search tree. Explain what is the running time of a dynamic programming implementation of this recursive formulation. The following figure illustrates these definitions.

Suppose that there are 4 items with weights $[2, 1, 3, 5]$ and number of accesses equal to $[1, 3, 2, 1]$. Below we show one possible choice of $T$:

![Tree showing item sizes](image)

The cost of a binary search tree is the sum of the costs of all its nodes. So the cost of the above example tree is 5. The task is to build a minimum cost binary search tree to store these $n$ items. Your algorithm should run in $O(n^3)$ time.

4. Optimal Merge Order. The input comprises a sequence of $n$ sorted lists $L_1, L_2, \ldots, L_n$. Let $l_i$ denote the length of list $L_i$. The task is to perform a series of $n - 1$ merges which will yield the merge of the $n$ input lists. At any stage, two adjacent lists are merged and
replaced in the list sequence by their merge. The cost for merging lists of lengths \( r \) and \( s \) is \( r + s \). So given an input \( L_1, L_2, L_3, L_4 \), with lengths 5, 3, 4, 4, respectively, we might first merge \( L_2 \) and \( L_3 \) producing list \( L_{23} \), and the new list sequence \( L_1, L_{23}, L_4 \), then merge \( L_{23} \) and \( L_4 \) producing list \( L_{24} \) and the list sequence \( L_1, L_{24} \) and then finally merge \( L_1 \) and \( L_{24} \). The cost would be 7 + 11 + 16 = 34. This is not the optimal choice.

The task is to find a minimum cost merge order. Give a recursive formulation of the function \( \text{Cost}(i, k) \), the minimum cost for merging the sequence of list \( L_i, L_{i+1}, \ldots, L_k \). Show how leads to an \( O(n^3) \) algorithm for finding a least cost merge order.

5. Consider the longest common subsequence problem for two strings \( X[1 : m] \) and \( Y[1 : n] \) in the situation that there are \( k \) different characters and a total of \( p \) different matching pairs, i.e. indices \( (i, j) \) such that \( X[i] = Y[j] \). You may assume that the characters are the integers \( 0, \ldots, k - 1 \). The problem is to find the length of a longest common subsequence of \( X \) and \( Y \) in \( O(n + m + pk) \) time.

The basic recursion is as usual, except now rather than delete a single character at a time, if considering a new rightmost character in \( X \), \( X[i] = c \) say, one considers the subproblem with the same rightmost character in \( Y \). Thus the idea is to generate at most roughly \( p \) recursive subproblems, and to consider up to \( k \) subproblems when making recursive calls.

In order to handle this efficiently, it will be helpful to form sorted lists of the occurrences of each character in each string.