1. Suppose you are given a DFA $M = (Q, \Sigma, \delta, s, F)$ and a binary string $w \in \Sigma^*$ where $\Sigma = \{0, 1\}$. Describe and analyze an algorithm that computes the longest subsequence of $w$ that is accepted by $M$, or correctly reports that $M$ does not accept any subsequence of $w$.

2. Problem 6.21 in Dasgupta et al. on finding the minimum sized vertex cover in a tree.

3. The McKing chain wants to open several restaurants along Red street in Shampoo-Banana. The possible locations are at $L_1, L_2, \ldots, L_n$ where $L_i$ is at distance $m_i$ meters from the start of Red street. Assume the street is a straight line and the locations are in increasing order of distance from the starting point (thus $0 \leq m_1 < m_2 < \ldots < m_n$). McKing has collected some data indicating that opening a restaurant at location $L_i$ will yield a profit of $p_i$ independent of where the other restaurants are located. However, the city of Shampoo-Banana has a zoning law which requires that any two McKing locations should be $D$ or more meters apart. In addition McKing does not want to open more than $k$ restaurants due to budget constraints. Describe an algorithm that McKing can use to figure out the maximum profit it can obtain by opening restaurants while satisfying the city’s zoning law and the constraint of opening at most $k$ restaurants. Your algorithm should use only $O(n)$ space.

4. Let $X = x_1, x_2, \ldots, x_r$, $Y = y_1, y_2, \ldots, y_s$ and $Z = z_1, z_2, \ldots, z_t$ be three sequences. A common supersequence of $X$, $Y$ and $Z$ is another sequence $W$ such that $X$, $Y$ and $Z$ are subsequences of $W$. Suppose $X = a, b, d, c$ and $Y = b, a, b, e, d$ and $Z = b, e, d, c$. A simple common supersequence of $X$, $Y$ and $Z$ is the concatenation of $X$, $Y$ and $Z$ which is $a, b, d, c, b, a, b, e, d, b, e, d, c$ and has length 13. A shorter one is $b, a, b, e, d, c$ which has length 6. Describe an efficient algorithm to compute the length of the shortest common supersequence of three given sequences $X$, $Y$ and $Z$.

5. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. Design an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in a given tree. See figure below for an example. Assume that the tree is binary (number of children is at most 2).

![A message being distributed through a tree in five rounds.](image)

**Solved Problems**

6. A string $w$ of parentheses ( and ) and brackets [ and ] is **balanced** if it is generated by the following context-free grammar:

$$S \rightarrow \varepsilon \mid (S) \mid [S] \mid SS$$

For example, the string $w = (())(())(())$ is balanced, because $w = xy$, where

$$x = (()) \quad \text{and} \quad y = (())(())$$.
Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets. Your input is an array $A[1 .. n]$, where $A[i] \in \{ (, [\} \}$ for every index $i$.


For all indices $i$ and $j$, let $LBS(i, j)$ denote the length of the longest balanced subsequence of the substring $A[i .. j]$. We need to compute $LBS(1, n)$. This function obeys the following recurrence:

$$LBS(i, j) = \begin{cases} 0 & \text{if } i \geq j \\ \max \left\{ 2 + LBS(i + 1, j - 1), \max_{k=1}^{j-1} (LBS(i, k) + LBS(k + 1, j)) \right\} & \text{if } A[i] \sim A[j] \\ \max_{k=1}^{j-1} (LBS(i, k) + LBS(k + 1, j)) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $LBS[1 .. n, 1 .. n]$. Since every entry $LBS[i, j]$ depends only on entries in later rows or earlier columns (or both), we can evaluate this array row-by-row from bottom up in the outer loop, scanning each row from left to right in the inner loop. The resulting algorithm runs in $O(n^3)$ time.

**LongestBalancedSubsequence($A[1 .. n]$):**

```python
for i ← n down to 1
    LBS[i, i] ← 0
for j ← i + 1 to n
    if A[i] \sim A[j]
        LBS[i, j] ← LBS[i + 1, j - 1] + 2
    else
        LBS[i, j] ← 0
    for k ← i to j - 1
        LBS[i, j] ← max {LBS[i, j], LBS[i, k] + LBS[k + 1, j]}
return LBS[1, n]
```

**Rubric:** 10 points, standard dynamic programming rubric

---

Oh, no! You've just been appointed as the new organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how fun the employee is. In order to keep things social, there is one restriction on the guest list: An employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company must attend the party, even though she has a negative fun rating; it’s her company, after all.

Describe an algorithm that makes a guest list for the party that maximizes the sum of the fun ratings of the guests. The input to your algorithm is a rooted tree $T$ describing the company hierarchy, where each node $v$ has a field $v.fun$ storing the fun rating of the corresponding employee.

**Solution:**[two functions] We define two functions over the nodes of $T$.

- $MaxFunYes(v)$ is the maximum total fun of a legal party among the descendants of $v$, where $v$ is definitely invited.
- $MaxFunNo(v)$ is the maximum total fun of a legal party among the descendants of $v$, where $v$ is definitely not invited.
We need to compute $\text{MaxFunYes}(\text{root})$. These two functions obey the following mutual recurrences:

\[
\text{MaxFunYes}(v) = v.\text{fun} + \sum_{\text{children } w \text{ of } v} \text{MaxFunNo}(w)
\]

\[
\text{MaxFunNo}(v) = \sum_{\text{children } w \text{ of } v} \max\{\text{MaxFunYes}(w), \text{MaxFunNo}(w)\}
\]

(These recurrences do not require separate base cases, because $\sum \emptyset = 0$.) We can memoize these functions by adding two additional fields $v.\text{yes}$ and $v.\text{no}$ to each node $v$ in the tree. The values at each node depend only on the values at its children, so we can compute all $2^n$ values using a post-order traversal of $T$.

\begin{mdframed}
\textbf{BestParty}(T):
\begin{algorithmic}
\State $\text{ComputeMaxFun}(T.\text{root})$
\Return $T.\text{root}.\text{yes}$
\end{algorithmic}
\end{mdframed}

\begin{mdframed}
\textbf{ComputeMaxFun}(v):
\begin{algorithmic}
\State $v.\text{yes} \leftarrow v.\text{fun}$
\State $v.\text{no} \leftarrow 0$
\For{all children $w$ of $v$}
\State $\text{ComputeMaxFun}(w)$
\State $v.\text{yes} \leftarrow v.\text{yes} + w.\text{no}$
\State $v.\text{no} \leftarrow v.\text{no} + \max\{w.\text{yes}, w.\text{no}\}$
\EndFor
\end{algorithmic}
\end{mdframed}

(Yes, this is still dynamic programming; were only traversing the tree recursively because thats the most natural way to traverse trees!\footnote{A naive recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio. The worst-case tree is a path every non-leaf node has exactly one child.}) The algorithm spends $O(1)$ time at each node, and therefore runs in $O(n)$ time altogether.
**Solution:** For each node \( v \) in the input tree \( T \), let \( \text{MaxFun}(v) \) denote the maximum total fun of a legal party among the descendants of \( v \), where \( v \) may or may not be invited.

The president of the company must be invited, so none of the presidents children in \( T \) can be invited. Thus, the value we need to compute is

\[
\text{root.fun} + \sum_{\text{grandchildren } w \text{ of root}} \text{MaxFun}(w).
\]

The function \( \text{MaxFun} \) obeys the following recurrence:

\[
\text{MaxFun}(v) = \max \left\{ \begin{array}{l} v.\text{fun} + \sum_{\text{grandchildren } x \text{ of } v} \text{MaxFun}(x) \\ \sum_{\text{children } w \text{ of } v} \text{MaxFun}(w) \end{array} \right\}
\]

(This recurrence does not require a separate base case, because \( \sum \emptyset = 0 \).) We can memoize this function by adding an additional field \( v.\text{maxFun} \) to each node \( v \) in the tree. The value at each node depends only on the values at its children and grandchildren, so we can compute all values using a postorder traversal of \( T \).

<table>
<thead>
<tr>
<th>BestParty(<em>T</em>):</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COMPUTE MAXFUN</strong>(<em>T</em>.root)</td>
</tr>
<tr>
<td>party ← <em>T</em>.root.fun</td>
</tr>
<tr>
<td>for all children <em>w</em> of <em>T</em>.root</td>
</tr>
<tr>
<td>for all children <em>x</em> of <em>w</em></td>
</tr>
<tr>
<td>party ← party + <em>x</em>.maxFun</td>
</tr>
<tr>
<td>return party</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ComputeMaxFun(<em>v</em>):</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes ← <em>v</em>.fun</td>
</tr>
<tr>
<td>no ← 0</td>
</tr>
<tr>
<td>for all children <em>w</em> of <em>v</em></td>
</tr>
<tr>
<td>ComputeMaxFun(<em>w</em>)</td>
</tr>
<tr>
<td>no ← no + <em>w</em>.maxFun</td>
</tr>
<tr>
<td>for all children <em>x</em> of <em>w</em></td>
</tr>
<tr>
<td>yes ← yes + <em>x</em>.maxFun</td>
</tr>
<tr>
<td><em>v</em>.maxFun ← max{yes, no}</td>
</tr>
</tbody>
</table>

(Yes, this is still dynamic programming; were only traversing the tree recursively because thats the most natural way to traverse trees\(^2\))

The algorithm spends \( O(1) \) time at each node (because each node has exactly one parent and one grandparent) and therefore runs in \( O(n) \) time altogether.

**Rubric:** 10 points: standard dynamic programming rubric. These are not the only correct solutions.

\(^2\) Like the previous solution, a direct recursive implementation would run in \( O(\phi^n) \) time in the worst case, where \( \phi = (1+\sqrt{5})/2 \approx 1.618 \) is the golden ratio.