1. Every year, as part of its annual meeting, the Antarctic Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to \( n \). During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

```
1 2 3 4 5 6 7 8
 8 1 5  2 6 3 4 7
```


For every pair of snails, the Antarctic SNUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array \( M[1..n, 1..n] \) posted on the wall behind the Round Table, where \( M[i, j] = M[j, i] \) is the reward to be paid if snails \( i \) and \( j \) meet. Rewards may be positive, negative, or zero.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array \( M \) as input.
2. Suppose you are given a NFA $M = (\{0, 1\}, Q, s, A, \delta)$ without $\epsilon$-transitions and a binary string $w \in \{0, 1\}^*$. Describe and analyze an efficient algorithm to determine whether $M$ accepts $w$. Concretely, the input NFA $M$ is represented as follows:

- $Q = \{1, 2, \ldots, k\}$ for some integer $k$.
- The start state $s$ is state 1.
- Accepting states are represented by a boolean array $Acc[1..k]$, where $Acc[q] = True$ if and only if $q \in A$.
- The transition function $\delta$ is represented by a boolean array $inDelta[1..k, \emptyset..1, 1..k]$, where $inDelta[p, a, q] = True$ if and only if $q \in \delta(p, a)$.

Your input consists of the integer $k$, the array $Acc[1..k]$, the array $inDelta[1..k, \emptyset..1, 1..k]$, and the input string $w[1..n]$. Your algorithm should return $true$ if $M$ accepts $w$, and $false$ if $M$ does not accept $w$. Report the running time of your algorithm as a function of $k$ (the number of states in $M$) and $n$ (the length of $w$). [Hint: Do not convert $M$ to a DFA!!]
Solved Problems

3. A string \( w \) of parentheses \((\) and \(\)) and brackets \([\) and \(\]) is balanced if and only if \( w \) 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 \( w = x y \), 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 \( i \).

**Solution:** Suppose \( A[1..n] \) is the input string. For all indices \( i \) and \( k \), let \( LBS(i,k) \) denote the length of the longest balanced subsequence of the substring \( A[i..k] \). We need to compute \( LBS(1,n) \). This function obeys the following recurrence:

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

Here \( A[i] \sim A[k] \) indicates that \( A[i] \) is a left delimiter and \( A[k] \) is the corresponding right delimiter: Either \( A[i] = ( \) and \( A[k] = ) \), or \( A[i] = [ \) and \( A[k] = ] \).

We can memoize this function into a two-dimensional array \( LBS[1..n,1..n] \). Because each 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.

```plaintext
LONGEST_BALANCED_SUBSEQUENCE(A[1..n]):
for i ← n down to 1
   LBS[i, i] ← 0
for k ← i + 1 to n
   if A[i] ∼ A[k]
      LBS[i, k] ← LBS[i + 1, k − 1] + 2
   else
      LBS[i, k] ← 0
   for j ← i to k − 1
      LBS[i, k] ← max{LBS[i, k], LBS[i, j] + LBS[j + 1, k]}
return LBS[1, n]
```

**Rubric:** 10 points, standard dynamic programming rubric
4. 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 \( v.\text{fun} \) 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 **MaxFunYes**\((\text{root})\). These two functions obey the following mutual recurrences:

\[
\begin{align*}
\text{MaxFunYes}(v) &= v.\text{fun} + \sum_{\text{children } w \text{ of } v} \text{MaxFunNo}(w) \\
\text{MaxFunNo}(v) &= \sum_{\text{children } w \text{ of } v} \max\{\text{MaxFunYes}(w), \text{MaxFunNo}(w)\}
\end{align*}
\]

(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 \( 2n \) values using a postorder traversal of \( T \).

(Yes, this is still dynamic programming; we’re only traversing the tree recursively because that’s the most natural way to traverse trees!\(^a\)) The algorithm spends \( O(1) \) time at each node, and therefore runs in \( O(n) \) time altogether.

---

\(^a\)A naïve 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.
Solution (one function): 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 president’s “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:

$$\text{MaxFun}(v) = \max \left\{ v.\text{fun} + \sum_{\text{grandchildren } x \text{ of } v} \text{MaxFun}(x), \sum_{\text{children } w \text{ of } v} \text{MaxFun}(w) \right\}$$

(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 $T$.

\begin{align*}
\text{BestParty}(T): \quad & \text{ComputeMaxFun}(T.\text{root}) \\
& \text{party} \leftarrow T.\text{root}.\text{fun} \\
& \text{for all children } w \text{ of } T.\text{root} \\
& \quad \text{for all children } x \text{ of } w \\
& \quad \quad \text{party} \leftarrow \text{party} + x.\text{maxFun} \\
& \text{return party}
\end{align*}

\begin{align*}
\text{ComputeMaxFun}(v): \quad & \text{yes} \leftarrow v.\text{fun} \\
& \text{no} \leftarrow 0 \\
& \text{for all children } w \text{ of } v \\
& \quad \text{ComputeMaxFun}(w) \\
& \quad \text{no} \leftarrow \text{no} + w.\text{maxFun} \\
& \quad \text{for all children } x \text{ of } w \\
& \quad \quad \text{yes} \leftarrow \text{yes} + x.\text{maxFun} \\
& \quad \text{v.\text{maxFun}} \leftarrow \max\{\text{yes, no}\}
\end{align*}

(Yes, this is still dynamic programming; we’re only traversing the tree recursively because that’s the most natural way to traverse trees!\(^a\))

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.

\(^a\)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.

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