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 $x = ( [ [ ) [ ] ( ) ] )$ and $y = [ ] ( () ) [ ]$.

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:**


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.

```plaintext
LongestBalancedSubsequence(A[1 .. n]):
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]
```
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 $MaxFunYes(root)$. These two functions obey the following mutual recurrences:

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

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

(These recurrences do not require separate base cases, because $\sum \emptyset = 0$.) We can memoize these functions by adding two additional fields $v.yes$ and $v.no$ 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 post-order traversal of $T$.

```plaintext
BestParty(T):
    ComputeMaxFun(T.root)
    return T.root.yes
```

```plaintext
ComputeMaxFun(v):
    v.yes ← v.fun
    v.no ← 0
    for all children w of v
        ComputeMaxFun(w)
        v.yes ← v.yes + w.no
        v.no ← v.no + max {w.yes, w.no}
```

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

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

(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 $T$.

Let $\text{BestParty}(T)$:

```plaintext
    COMPUTEMAXFUN(T.root)
    party ← T.root.fun
    for all children $w$ of $T.root$
        for all children $x$ of $w$
            party ← party + $x.\text{maxFun}$
    return party
```

Let $\text{ComputeMaxFun}(v)$:

```plaintext
    yes ← v.fun
    no ← 0
    for all children $w$ of $v$
        COMPUTEMAXFUN(w)
        no ← no + $w.\text{maxFun}$
    for all children $x$ of $w$
        yes ← yes + $x.\text{maxFun}$
    v.\text{maxFun} ← \max \{yes, no\}
```

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

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.