Standard 10-point dynamic programming rubric. As usual, a score of x on the following 10-point scale corresponds to a score of $\lceil x/3 \rceil$ on the 4-point homework scale.

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit, unless the problem says otherwise.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but iterative psuedocode is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, memoization structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

  We rarely include these target time bounds in the actual questions, because when we have included them, significantly more students turned in algorithms that meet the target time bound but didn't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

1. Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets.

Solution: Suppose $S[1..n]$ is the input string. For all indices i and j, we write $\boldsymbol{S[i] \sim S[j]}$ to indicate that $S[i]$ and $S[j]$ are matching delimiters: either $S[i] =$ **(** and $S[j] =$ **)** or $S[i] =$ **[** and $S[j] =$ **]**.

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

$$LBS(i,j) = \begin{cases} 0 & \text{if } i \geq j \\ \max\left\{ \begin{array}{c} 2 + LBS(i+1, j-1) \\ \max\left\{ LBS(i,k) + LBS(k+1,j) \;\middle|\; i \leq k < j \right\} \end{array} \right\} & \text{if } S[j] \sim S[j] \\ \max\left\{ LBS(i,k) + LBS(k+1,j) \;\middle|\; i \leq k < j \right\} & \text{otherwise} \end{cases}$$

In fact, we can simplify this recurrence slightly using a greedy exchange argument:

$$LBS(i,j) = \begin{cases} 0 & \text{if } i \geq j \\ 2 + LBS(i+1, j-1) & \text{if } S[j] \sim S[j] \\ \max\left\{ LBS(i,k) + LBS(k+1,j) \;\middle|\; i \leq k < j \right\} & \text{otherwise} \end{cases}$$

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 $\boldsymbol{O(n^3)}$ ***time***.

```
LongestBalancedSubsequence(S[1..n]):
  for i ← n down to 1
      LBS[i,i] ← 0
  for j ← i+1 to n
      if S[i] ~ S[j]
          LBS[i,j] ← LBS[i+1,j-1] + 2
      else
          LBS[i,j] ← 0
      for k ← i to j-1
          LBS[i,j] ← min{LBS[i,j], LBS[i,k] + LBS[k+1,j]}
  return LBS[1,n]
```

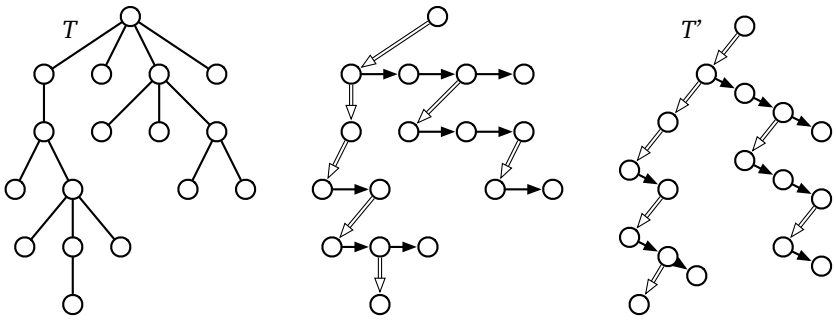
■

2. Describe an algorithm that finds the set of k employees to invite that maximizes the sum of the k resulting "fun" values. The input to your algorithm is the tree T, the integer k, and the *With* and *Without* values for each employee.

Solution: We assume that the children of every node in T are arbitrarily ordered from left to right, and that every node v in the tree T has pointers to three other nodes:

- $parent(v)$ — the parent of v (or NULL if v is the root)
- $first(v)$ — the first child of v (or NULL if v is a leaf)
- $next(v)$ — the child of $parent(v)$ just after v (or NULL if v is the last child of $parent(v)$)

The *first* and *next* pointers implicitly define a binary tree T'; specifically, $first(v)$ and $next(v)$ are the left and child of v in T'. These are very different trees—in particular, the same node may have different parents in T and T'—but we will implicitly use the recursive structure of T' to guide any recursively defined function of T.



Representing an arbitrary rooted tree T using *first* and *next* pointers, and the equivalent binary tree T'.

We say that a node v is a ***follower*** of node u in T if there is a path in T from u to v following only *first* and *next* pointers. Equivalently, the followers of u in T are precisely the descendants of u in the binary tree T'.

For every node v and every non-negative integer k we define two values:

- $FunY(v, \ell)$ — The maximum total fun achieved by inviting exactly ℓ followers of v, given that $parent(v)$ is also invited.
- $FunN(v, \ell)$ — The maximum total fun achieved by inviting exactly ℓ followers of v, given that $parent(v)$ is *not* also invited.

We need to compute $FunY(root, k)$.

These two functions obey the following mutual recurrence. In the most general case, we have to decide (1) whether or not to invite v and (2) how to distribute the remaining invitations between the followers of $first(v)$ and the followers of $next(v)$.

$$FunY(v,\ell) = \begin{cases} 0 & \text{if } \ell = 0 \\ -\infty & \text{if } \ell > 0 \text{ and } v = \text{NULL} \\ \max\left\{ \begin{array}{l} \displaystyle\max_{1\le j\le \ell} \left(\begin{array}{r} with(v) + FunY(first(v), j-1) \\ + FunY(next(v), \ell - j) \end{array} \right) \\ \displaystyle\max_{0\le j\le \ell} \left(\begin{array}{r} FunN(first(v), j) \\ + FunY(next(v), \ell - j) \end{array} \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$FunN(v,\ell) = \begin{cases} 0 & \text{if } \ell = 0 \\ -\infty & \text{if } \ell > 0 \text{ and } v = \textsc{Null} \\ \max\left\{ \begin{array}{l} \max\limits_{1\le j\le \ell} \left(\begin{array}{c} with(v) + FunY(first(v), j-1) \\ + FunN(next(v), \ell - j) \end{array} \right) \\ \max\limits_{0\le j\le \ell} \left(\begin{array}{c} FunN(first(v), j) \\ + FunN(next(v), \ell - j) \end{array} \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize these functions by storing two one-dimensional arrays $FunY[1..k]$ and $FunN[1..k]$ at every node v. Alternatively, we can arbitrarily index the vertices of T from 1 to n and then memoize into a pair of two-dimensional arrays $FunY[1..n, 1..k]$ and $FunN[1..n, 1..k]$.

The function values for any node v depend only on the values for $first(v)$ and $next(v)$. Thus, we can fill the memoization structure using a right-to-left postorder traversal of T in the outer loop—or equivalently, a postorder traversal of the binary tree T'—and considering all possible values of ℓ in arbitrary order in the middle loop. For each pair (v, ℓ), we need a third inner loop to compute $FunY[v, \ell]$ and $FunN[v, \ell]$, in $O(\ell) = O(k)$ time.

Altogether, the resulting dynamic programming algorithm runs in $\boldsymbol{O(nk^2)}$ ***time*** and uses $\boldsymbol{O(nk)}$ ***space***. It is impossible to invite more than n people, so we can safely assume that $k \le n$. Thus, the algorithm runs in at most $\boldsymbol{O(n^3)}$ ***time*** and uses at most $\boldsymbol{O(n^2)}$ ***space***.

This algorithm can be improved to run in only $O(kn \log n) = O(n^2 \log n)$ time, using only $O(k \log n) = O(n \log n)$ space (in addition to the input tree T), with a technique called *heavy path decomposition* that we won't discuss this semester. ■

Rubric: Standard dynamic programming rubric. An algorithm that works correctly only for binary trees is worth up to 3 points; scale partial credit.

3. Describe and analyze an algorithm to compute the *number* of shortest paths from a source vertex s to a target vertex t in a directed graph G with weighted edges.

Solution: We start by computing shortest-path distances $dist(v)$ from s to v, for every vertex v, using Dijkstra's algorithm. Call an edge $u \to v$ ***tight*** if $dist(u) + w(u \to v) = dist(v)$. Every edge in a shortest path from s to t must be tight; conversely, every path from s to t that uses only tight edges is in fact a shortest path.

Let H be the subgraph of tight edges in G. We can easily construct H in $O(V + E)$ time. Because all edge weights are positive, H is a directed acyclic graph. It remains only to count the number of paths from s to t in H.

For any vertex v, let $PathsToT(v)$ denote the number of paths in H from v to t; we need to compute $PathsToT(s)$. This function satisfies the following recurrence:

$$PathsToT(v) = \begin{cases} 1 & \text{if } v = t \\ \displaystyle\sum_{v \to w} PathsToT(w) & \text{otherwise} \end{cases}$$

In particular, the base case where v is a sink (and therefore has no paths to t) is correctly handled implicitly by the empty sum.

We can memoize this function into the graph itself, storing each value $PathsToT(v)$ at the correspinding vertex v. Since each subproblem depends only on its successors in H, we can compute $PathsToT(v)$ for all vertices v by considering the vertices in reverse topological order, or equivalently, by performing a depth-first search of H starting at s. The resulting algorithm runs in $O(V + E)$ time.

The overall running time of the algorithm is dominated by Dijkstra's algorithm in the preprocessing phase, which runs in $\boldsymbol{O(E \log V)}$ ***time***.

Actually implementing this algorithm requires more care. The number of shortest paths can be exponential in V, so just storing the exact number of shortest paths might require $\Omega(V)$ bits. Thus, we cannot use standard integer type to store $PathsToT(v)$; in particular, each of the additions used to compute $PathsToT(v)$ requires $\Theta(V)$ time in the worst case. The worst-case running time of our algorithm is actually $O(EV \log V)$ if we count bit operations, or $O(EV)$ if we can add $\log V$-bit integers in constant time. ■