Describe and analyze a dynamic programming algorithm to construct an optimal AVL tree for a given set of search keys and frequencies. An algorithm that computes the cost of the optimal AVL tree is worth full credit.

**Solution:** We'll use a similar solution to the regular optimal binary search tree problem, but along with building optimal AVL trees for every subset of contiguous keys, we'll also set requirements on their heights so it's easier to enforce that we're building AVL trees.

Let $OptAVL(i, k, h)$ be the optimal cost of an AVL tree for the subarray $A[i .. k]$ that has height exactly $h$ or $\infty$ if no AVL tree of height $h$ exists. For simplicity, we'll say an AVL tree over zero keys has height $-1$. The optimal AVL tree for $A[i .. k]$ of height $h$ has at least one subtree of height exactly $h - 1$. The other subtree has a height $h - 2$ or $h - 1$. Like in the original problem, if the optimal AVL tree has root $A[r]$, then its two subtrees are optimal AVL tree of a particular height over $A[i .. r - 1]$ and $A[r + 1 .. k]$, and the total cost is the cost of searching within those trees plus the sum of frequencies for $A[i .. k]$. As before, we can precompute an array $F[i, k]$ such that $F[i, k] = \sum_{j=i}^{k} f[j]$ in $O(n^2)$ time using the procedure IntPt$f[1 .. n])$ given in Erickson 3.9. We derive the following recurrence for $OptAVL(i, k, h)$.

$$OptAVL(i, k, h) = \begin{cases} 0 & \text{if } i > k \text{ and } h = -1 \\ \infty & \text{if } i > k \text{ and } h \neq -1 \\ F[i, k] & \text{otherwise} \\ + \min_{i \leq r \leq k} \min \left\{ \begin{array}{l} OptAVL(i, r - 1, h - 1) \\ + OptAVL(r + 1, k, h - 2), \\ OptAVL(i, r - 1, h - 1) \\ + OptAVL(r + 1, k, h - 1), \\ OptAVL(i, r - 1, h - 2) \\ + OptAVL(r + 1, k, h - 1) \end{array} \right\} \end{cases}$$

Each subproblem can be specified by three integers $i$, $k$, and $h$ such that $1 \leq i \leq n + 1$, $0 \leq k \leq n$, and $-1 \leq h \leq n - 1$, so we can store all possible values of $OptAVL$ in a three-dimensional array $OptAVL[1 .. n + 1, 0 .. n, -1 .. n - 1]$. Each entry depends upon entires with higher $i$ value or lower $k$ value and strictly smaller $h$ value. Therefore, we can fill the array by increasing $h$ in the outer loop, increasing $j$ within that loop, and decreasing $i$ within that loop. There are $O(n^3)$ entries to fill, and it takes $O(n)$ time to fill each by trying all possible $r$ values. Therefore, the running time will be $O(n^4)$. We need to return the smallest $OptAVL[1, n, h]$ over all choices of $h$ from $-1$ to $n - 1$.

Here is a procedure `ComputeOptAVL(i, k, h)` that fills the entry $OptAVL[i, k, h]$ assuming the entries it depends upon have already been computed.
And here is a procedure \texttt{OptimalAVL}($f[1..n]$) that computes the cost of the optimal AVL tree for the given frequencies.

\begin{verbatim}
\textbf{OptimalAVL}(f[1..n]):
    \textbf{InitF}(f[1..n])
    for k ← 0 to n
        OptAVL[k+1, k, -1] ← 0
    for h ← 0 to n-1
        for k ← 0 to n
            OptAVL[k+1, k, h] ← ∞
            for i ← k down to 1
                \textbf{computeOptAVL}(i, k, h)
        best ← ∞
        for h ← -1 to n
            if best > OptAVL[1, n, h]
                best ← OptAVL[1, n, h]
    return best
\end{verbatim}

Finally, a few students observed that since we’re building an AVL tree, it must have height at most $O(\log n)$. We can improve the running time to $O(n^3 \log n)$ by only consider $h$ between $-1$ and $c \log n$ for some sufficiently large value $c$. \hfill \blacksquare

\textbf{Rubric:} 10 points total: 5 points total for the recurrence; -2 for no justification, -1 for missing base case(s); 3 points for filling the table (0 if the recurrence is very wrong); 2 points for running time analysis. +3 points extra credit for a correct $O(n^3 \log n)$ time solution.
Describe and analyze a dynamic programming algorithm to distribute gifts so that the minimum number of people get fired.

**Solution:** Like the maximum independent set problem on trees, we want to recursively compute solutions on rooted subtrees. However, the gifts available for the root of a given subtree and the cost of giving particular gifts depend upon what gift their parent receives. To keep things relatively simple, then, we’ll take inspiration from the second solution in Erickson 3.10 by creating subproblems that specify what gift the subtree root must take.

Let \( \text{PartyCost}(v, g) \) denote the minimum cost of assigning gifts to the subtree rooted at \( v \) given that \( v \) is assigned gift \( g \in \{1, 2, 3\} \). An optimal solution assigns one of the two gifts not equal to \( g \) to each of the children of \( g \), incurring a cost of 1 per child that receives a lower numbered gift. Subject to the choices of gifts for each child, we want the best labeling for their respective subtrees, and then we want the gift choices that minimizes the total cost of possibly firing the child and optimizing for their subtree. Let \( w \downarrow v \) mean “\( w \) is a child of \( v \)”. We have the following recurrence for \( \text{PartyCost}(v, g) \).

\[
\text{PartyCost}(v, g) = \begin{cases} 
\sum_{w \downarrow v} \min \{\text{PartyCost}(w, 2), \text{PartyCost}(w, 3)\} & \text{if } g = 1 \\
\sum_{w \downarrow v} \min \{1 + \text{PartyCost}(w, 1), \text{PartyCost}(w, 3)\} & \text{if } g = 2 \\
\sum_{w \downarrow v} \min \{1 + \text{PartyCost}(w, 1), 1 + \text{PartyCost}(w, 2)\} & \text{otherwise}
\end{cases}
\]

If \( r \) is the root of \( T \), we want to compute \( \min_{1 \leq g \leq 3} \{\text{PartyCost}(r, g)\} \).

For each node \( v \), we can store \( \text{PartyCost}(v, g) \) in a three element array \( v.\text{PartyCost}[1..3] \) where \( v.\text{PartyCost}[g] = \text{PartyCost}(v, g) \). Each entry \( v.\text{PartyCost}[g] \) depends upon children node of \( v \), so we can fill the entries in postorder. Each vertex contributes a constant number of values to its parent so we spend \( O(n) \) time performing the postorder traversal of the tree and computing entries in the arrays. Here’s a procedure \( \text{GIVEGIFTS}(T) \) which computes the minimum number of firings for the party.

```plaintext
GIVEGIFTS(T):
for each node v of T in postorder
    v.\text{PartyCost}[1] ← 0
    v.\text{PartyCost}[2] ← 0
    v.\text{PartyCost}[3] ← 0
for each child w of v
    v.\text{PartyCost}[1] ← v.\text{PartyCost}[1] + \min\{w.\text{PartyCost}[2], w.\text{PartyCost}[3]\}
    v.\text{PartyCost}[2] ← v.\text{PartyCost}[2] + \min\{1 + w.\text{PartyCost}[1], w.\text{PartyCost}[3]\}
    v.\text{PartyCost}[3] ← v.\text{PartyCost}[3] + \min\{1 + w.\text{PartyCost}[1], 1 + w.\text{PartyCost}[2]\}

r ← root of T
return \min\{r.\text{PartyCost}[1], r.\text{PartyCost}[2], r.\text{PartyCost}[3]\}
```

**Rubric:** 10 points total: 5 points total for the recurrence; -2 for no justification, -1 for missing base case(s); 3 points for memoization (0 if the recurrence is very wrong); 2 points for running time analysis.
Describe and analyze an algorithm to compute the \textit{minimum} number of steps required to produce any given integer \( n \).

**Solution:** We’ll use the following greedy algorithm: If \( n = 1 \), we do nothing and return 0. Otherwise, if the target value \( n \) is even, then we end with a doubling and find the minimum number of steps to compute \( n/2 \). Otherwise otherwise, we end with an increment (by necessity) and find the minimum number of steps to compute \( n - 1 \). We will prove our greedily chosen last step for even \( n \) belongs to some optimal solution. By induction on \( n \), our algorithm will then find the minimum number of steps needed to prepare for the final step.

Consider some minimum length sequence of increments and doublings \( S \) that start at 1 and ends at \( n \) where \( n \) is even. We may assume the first operation is a doubling, because incrementing and doubling 1 both lead to 2. Let \( x \) be the largest value doubled so that \( S \) ends with a subsequence \( S' \) that doubles \( x \) and then performs only increments to reach \( n \). We have \( x \leq n/2 \), and \( S' \) has length \( n - 2x + 1 \). Exchange subsequence \( S' \) with one that does increments from \( x \) to \( n/2 \) and then does one doubling to reach \( n \). The new subsequence has length \( n/2 - x + 1 \leq n - 2x + 1 \). After the exchange, we have a new sequence that uses at most as many steps as \( S \) while ending with a doubling as desired.

We can make the above algorithm iterative by just applying our greedy rule repeatedly and counting how many steps we use. The procedure \texttt{CountSteps}(\( n \)) returns the minimum number of steps to change 1 into \( n \).

```python
CountSteps(n):
    count ← 0
    x ← n
    while x ≠ 1
        count ← count + 1
        if x is even
            x ← x/2
        else
            x ← x - 1
    return count
```

We can half \( n \) at most \( O(\log n) \) times, and there are at most as many decrements as halvings. The algorithm runs in \( O(\log n) \) time.

**Rubric:** 10 points total: 3 points for the algorithm; 1 point for running time analysis; 6 points for justification.

A correct dynamic programming algorithm running in \( O(n) \) time is worth 5 points.
Describe an algorithm to find all vertices in $G$ that can be reached from a given vertex $v$ through a French flag walk.

**Solution:** Per the hint, we'll perform a reduction to normal graph reachability that uses a different graph $G'$. We need to know how many steps $\ell \mod 3$ it takes to reach each vertex in $G$, so we'll make three copies of each vertex and set up edges so that the copy you reach tells you how many steps you took to reach it and you can only take correctly colored edges based on that number of steps.

The procedure $\text{FFReachable}(V, E, v)$ takes the vertices and colored edges of $G$ along with a particular vertex $v$. It returns an array containing all vertices reachable from $v$ through a French flag walk.

\[
\begin{align*}
\text{FFReachable}(V, E, v): \\
(V', E') &\leftarrow (\emptyset, \emptyset) \\
\text{for each vertex } u \in V &\text{ add } (u, 0), (u, 1), \text{ and } (u, 2) \text{ to } V' \\
\text{for each edge } u \rightarrow w \in E &\text{ if } u \rightarrow w \text{ is red add } (u, 0) \rightarrow (w, 1) \text{ to } E' \\
&\text{ if } u \rightarrow w \text{ is white add } (u, 1) \rightarrow (w, 2) \text{ to } E' \\
&\text{ if } u \rightarrow w \text{ is blue add } (u, 2) \rightarrow (w, 0) \text{ to } E' \\
\text{Perform a BFS on } (V', E') \text{ starting from } (v, 0) &\text{ count } \leftarrow 0 \\
\text{for each vertex } u \in V &\text{ if } (u, 0), (u, 1), \text{ or } (u, 2) \text{ is marked add } \text{count} \leftarrow \text{count} + 1 \text{ reachable}[\text{count}] \leftarrow u \\
\text{return reachable}
\end{align*}
\]

We need to prove at least one of $(u, 0), (u, 1), \text{ or } (u, 2)$ is reachable from $(v, 0)$ in $G'$ if and only if $u$ is reachable from $v$ in $G$ through a French flag walk.

Suppose $u$ is reachable from $v$ in $G$ through a French flag walk of length $\ell$. We'll prove $(u, \ell \mod 3)$ is reachable from $(v, 0)$ using induction on $\ell$. If $\ell = 0$, then $u = v$ and indeed $(v, 0)$ can reach itself. Otherwise, the French flag walk begins with a walk of length $\ell - 1$ to some vertex $w$ followed by an edge $w \rightarrow u$. Vertex $(w, \ell - 1 \mod 3)$ is reachable from $(v, 0)$ by induction, and we added the edge $(w, \ell - 1 \mod 3) \rightarrow (u, \ell \mod 3)$ while creating $G'$, meaning $(u, \ell \mod 3)$ is reachable as well.

Now suppose $(u, i)$ is reachable from $(v, 0)$ in $G'$ for some $i \in \{0, 1, 2\}$ using a walk of length $\ell$. We'll prove $\ell = i \mod 3$ and that $u$ is reachable from $v$ using a French flag walk with $\ell$ edges again using induction on $\ell$. If $\ell = 0$, then $(u, i) = (v, 0)$ and indeed $v$ can reach itself. Otherwise, the walk to $(u, i)$ begins with a walk of length $\ell - 1$ ending at some vertex $(w, \ell - 1 \mod 3)$ by induction. Also, there is a French flag walk from $v$ to $w$ of length $\ell - 1$. By construction, the last edge on the walk in $G'$ must go to $(u, \ell \mod 3)$, and it has the correct color for extending the walk in $G$ from $w$ to $u$. 

1
Creating $G'$ and performing the BFS takes $O(V + E)$ time. It takes only $O(V)$ time to see which vertices are reachable from $(v, 0)$ using the marks, so the algorithm runs in $O(V + E)$ time total.

Rubric: 10 points total: 5 points for the algorithm; 3 points for justification; 2 points for running time analysis. This justification is more detailed than necessary for full credit, but there should be some effort made toward an if and only if argument.
Let $G$ be a directed acyclic graph whose vertices have labels over some fixed alphabet, and let $A[1 \ldots \ell]$ be a string over the same alphabet. Any directed path in $G$ has a label, which is a string obtained by concatenating the labels of its vertices.

**(a)** Describe and analyze a dynamic programming algorithm that correctly determines if there is a path in $G$ whose label is $A$.

**Solution:** We’ll take inspiration from the longest path example that a path begins with a single vertex and ends with another path. However, that other path must be labeled with a suffix of $A$, because the first character was already covered by the first vertex of the path. We’ll have to have subproblems based on which vertex the path starts with and what suffix of $A$ we’re working with.

For any vertex $v$ and integer $i$ where $1 \leq i \leq n$, let $\text{ContainsString}(v, i)$ be a boolean that is True if and only if there is a path in $G$ starting at $v$ whose label is $A[i .. n]$. Let $v.L$ denote the label of $v$. For $\text{ContainsString}(v, i)$ to be True, we need $v.L = A[i]$. If $i = n$, then that suffices as well. If $i < n$, then it is also necessary (but sufficient) for there to be at least one edge $v \rightarrow w$ where there is a path beginning at $w$ with the label $A[i + 1 .. n]$. We have the following recurrence for $\text{ContainsString}(v, i)$.

$$\text{ContainsString}(v, i) = \begin{cases} [v.L = A[i]] & \text{if } i = n \\ [v.L = A[i]] \land \left( \bigvee_{v \rightarrow w} \text{ContainsString}(w, i + 1) \right) & \text{otherwise} \end{cases}$$

Our goal is to see if there exists some vertex $v$ where $\text{ContainsString}(v, 1)$ is True.

We can memoize this recurrence by storing solutions in one array $v.\text{ContainsString}[1 .. n]$ per vertex $v$ where $v.\text{ContainsString}[i] = \text{ContainsString}(v, i)$. Each entry $v.\text{ContainsString}[i]$ depends upon entries on outgoing neighbors of $v$ and with larger $i$ value. Because $G$ is a DAG, we can evaluate entries by decreasing $i$ index in the outer loop and in postorder in an inner loop. Within each of the $n$ iterations of the outer loop, we’ll use each edge exactly once to do evaluations and do one evaluation for each vertex, so the whole algorithm will run in $O(n(V + E))$ time.

The procedure $\text{FINDSTRING}(V, E, A[1 .. n])$ takes a labeled DAG $G = (V, E)$ and returns True if and only if there is a path in $G$ with label $A$. 
(b) Describe and analyze a dynamic programming algorithm to find the number of paths in $G$ whose label is $A$. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

**Solution:** This solution requires just a few changes to the previous one.

For any vertex $v$ and integer $i$ where $1 \leq i \leq n$, let $NumberStrings(v, i)$ be the number of paths in $G$ starting at $v$ whose label is $A[i..n]$. If $v.L \neq A[i]$, then $NumberStrings(v, i) = 0$. If $v.L = A[i]$ and $i = n$, then $NumberStrings(v, i) = 1$. Otherwise, we're counting paths that start at $v$, take some edge $v \rightarrow w$ and then continue on. We want the total number of all such paths across all choices of $w$, so we should sum over the $NumberStrings(w, i + 1)$ values. We have the following recurrence for $NumberStrings(v, i)$.

$$NumberStrings(v, i) = \begin{cases} 
0 & \text{if } v.L \neq A[i] \\
1 & \text{if } v.L = A[i] \text{ and } i = n \\
\sum_{v \rightarrow w} NumberStrings(w, i + 1) & \text{otherwise}
\end{cases}$$

Paths could begin at any vertex, so our goal is to compute $\sum_v NumberStrings(v, 1)$.

As before, we can memoize this recurrence by storing solutions in one array $v.NumberStrings[1..n]$ per vertex $v$ where $v.NumberStrings[i] = NumberStrings(v, i)$. The dependencies, evaluation order, time per subproblem, and therefore total running time of $O(n(V + E))$ are the same as in part (a).

The procedure $CountPaths(V, E, A[1..n])$ takes a labeled DAG $G = (V, E)$ and returns the number of paths in $G$ whose label is $A$. 

---

**Rubric:** 5 points total: 2.5 points total for the recurrence; -1 for no justification, -0.5 for missing base case(s); 1.5 points for filling the table (0 if the recurrence is very wrong); 1 point for running time analysis.
Rubric: 5 points total: 2.5 points total for the recurrence; -1 for no justification, -0.5 for missing base case(s); 1.5 points for filling the table (0 if the recurrence is very wrong); 1 point for running time analysis.