Solution Format for Dynamic Programming

In general, we expect a dynamic programming solution to consist of 3 parts. First, it must clearly identify what the subproblems are, and include a short justification that the problems exhibit the optimal substructure property, if this is not obvious. Second, the recurrence relation between subproblems must be stated, including the base cases. For the recurrence relation to be valid, there must be no cyclic dependencies, and if it is not obvious (and we are using an iterative solution), the topological ordering must be stated. Finally, there must be a runtime analysis.

These three steps hold necessary whether we are using a recursive (top-down) strategy or an iterative (bottom-up) strategy.

Longest Path in a DAG

Suppose we wish to find the longest simple path (not repeating vertices) in a connected DAG with positive weights. Note that the general longest path problem for an arbitrary graph is not very easy to tackle using dynamic programming, because it lacks the optimal substructure property. For example, consider a complete graph on three nodes A, B, C; then the longest path from A to C is A, B, C, yet the longest path from A to B is not just A, B, so the substructure is violated.

Luckily, a positive edge weight connected DAG does have the optimal substructure property. Note that if we topologically order the vertices in the DAG, the longest path must start from the source s, the first node in the topological sort. To see this, note that by connectedness, if the longest path were from u to v, we can make the path strictly longer by including any path from s to u. By similar logic, it’s clear that the path must end on t, the last vertex in our topological sort.

Now, the subproblems in our dynamic programming example will be \( d[v] \) for each vertex v, defined to be the longest (greatest weight) path from s to v. Then, the recursion for our dynamic program setup is as follows: we consider each incoming edge u, v to a given vertex v, as the last edge in the greatest weight path from s to v. Then, assuming this edge u, v is the last edge, the greatest weight path has weight \( d[u] + w(u, v) \). Thus, the recursion is \( d[v] = \max_{(u, v) \in E} (d[u] + w(u, v)) \). The iterative DP solution here is straightforward: we consider each vertex iteratively in the topological sort order, because in this setup, edges in the DAG reversed correspond to dependencies in the DP problem.

Pseudocode for the algorithm is as follows.
DAG-LONGEST-PATH(V, E)
1 ordering = TOPOSORT(V, E)
2 Initialize d empty dictionary
3 for v ∈ ordering
4 d[v] = −∞
5 d[ordering[1]] = 0
6 for v ∈ ordering
7 for u, v ∈ E
8 d[v] = max(d[v], d[u] + w(u, v))
9 return d[ordering[−1]]

We can perform a complexity analysis. The topological sort takes $O(V + E)$, and each case considered in lines 6 and 7 takes $O(1)$ due to memoization, so these also take $O(V + E)$. Note that the resulting complexity is the same as that of the algorithm DAG-SP. This should be intuitively clear, because this is essentially the same problem but with largest weight instead of shortest weight (we can negate all weights and run DAG-SP and get the same result). Because DAG-SP has the optimal substructure property as well, it is essentially a DP algorithm.

**Rehash: Shortest Paths Dynamic Programming**

Recall from lecture our recursive solution to the single source shortest paths problem for non-negative-weight-cycle graphs. We created subproblems indexed on $v, k$ defined to be the shortest path from $s$ to $v$ using at most $k$ edges. Then, using the recursion we defined, $d[v, k] = \min(min_{u,v∈E}(d[u, k - 1] + w(u, v)), d[v, k - 1])$, with the base cases $d[s, 0] = ∞$ for all $v$ except $d[s, 0] = 0$. Notice that this is essentially the same approach as Bellman-Ford, which has the property that upon $k$ iterations of relaxing all edges, we have updated the distance to optimize paths on $k$ edges or less. For a similar reasoning, we only need to consider subproblems $v, k$ for $k$ in the range $0, |V| - 1$, because any longer paths would necessarily repeat a vertex by pigeonhole, causing a non-negative weight cycle.

We provide the following pseudocode for this algorithm.

SHORTEST-PATHS-DP(V, E, s)
1 Initialize d empty dictionary
2 for v ∈ ordering
3 d[v, 0] = ∞
4 d[s, 0] = 0
5 for i ∈ [1, |V| - 1]
6 for v ∈ V
7 d[v, i] = d[v, i - 1]
8 for u, v ∈ E
9 d[v, i] = max(d[v, i], d[u, i - 1] + w(u, v))
10 return d
The complexity of this algorithm is the same as Bellman-Ford, in that it relaxes each of \( E \) edges \( O(V) \) times for a runtime of \( O(VE) \).

**Practice DP problem: Crazy 8’s**

In the game Crazy 8’s, we are given an input of a sequence of cards \( C[0], \ldots, C[n-1] \), e.g., 7♣, 7♥, K♣, K♠, 8♥. We want to find the longest subsequence of cards where consecutive cards must have the same value, same suit, or have one of the two cards be an eight. The longest such subsequence in the example is 7♣, K♣, K♠, 8♥.

To solve this, if the cards are stored in array \( C \), we will keep an auxiliary score array \( S \) where \( S[i] \) represents the length of the longest subsequence ending with card \( C[i] \).

We start with \( S[0] = 1 \) since the longest subsequence ending with the first card is that card itself and has a length of 1. We iteratively calculate the next score \( S[i] \) by scanning all previous scores and set \( S[i] \) to be \( S[k] + 1 \) where \( S[k] \) represents the length of the longest subsequence that card \( C[i] \) can be appended to.

**Analysis**  For an input of \( n \) cards, there are \( O(n) \) subproblems: \( S[0], \ldots, S[n-1] \). Solving each subproblem requires iterating over all previous subproblems, for an \( O(n) \) time per subproblem. Thus in total, our runtime is \( O(n^2) \).

**Practice DP problem: Squeaky the Squirrel**

Squeaky the Squirrel comes across a large walnut tree containing \( n \) branch points (nodes). Each node \( v \) in the tree contains \( h(v) \) nuts. Every walnut increases his Happiness by 1 unit. However, each node also contains \( g(v) \) thorns that Squeaky has to sweep off the tree with his feet. Every such thorn decreases his Happiness by 1 unit. Happiness is additive; if Squeaky chooses to visit node \( v \), then his net Happiness gain would be \( h(v) - g(v) \).

You are given the tree \( T = (V, E) \) and the functions \( h(v), g(v) \). You want to calculate the maximum net Happiness gained by Squeaky when he begins at the root. You correctly deduce that this problem can be solved using structural DP. To solve it, you initialize a table \( DP[v] \) that contains the Happiness of the optimal subtree rooted at node \( v \).

If we define \( DP[v] \) so that you must include node \( v \), we have:

\[
DP[v] = h(v) - g(v) + \sum_{u \in v.\text{children}} \max(DP[u], 0).
\]

If \( v \) has no children, it follows that \( DP[v] = h(v) - g(v) \). In this case, the maximum net Happiness is just \( \max(DP[\text{root}], 0) \).

Alternatively, if \( DP[v] \) does not imply we include node \( v \), we have:

\[
DP[v] = \max(0, (h(v) - g(v) + \sum_{u \in v.\text{children}} DP[u]))
\]
If $v$ has no children, it follows that $DP[v] = \max(0, h(v) - g(v))$. The maximum net Happiness in this case is simply $DP[\text{root}]$.

**Analysis**  There are $O(n)$ subproblems (one for each node), and for each node $v$, the subproblem takes $O(\text{degree}(v))$. $\sum_{v \in V} O(\text{degree}(v)) = O(n)$, so the total runtime is $O(n)$. 