Dynammic Programming

Main Focus: Introduction

Dynammic Programming: Dynamic programming is method to quickly solve large problems by first solving intermediate problems, then using these intermediate problems to solve the large problem.

Bottom-Up DP: We evaluate the function starting with the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values, we store them in a table for memory purposes. As larger arguments are evaluated, pre-computed values for arguments can be used.

Top-Down DP: In this, the problem is broken down into sub problems. Each of these sub problems are solved and solution is remembered (memory), in case they need to be solved. We save each computed value as the final recursive function, and as the first action we check if pre-computed value exists.

Dynamic Programming Strategy

Dynamic programming and memoization work together. The main difference between dynamic programming and
divide and conquer is that in the case of the latter, sub problems are independent, whereas in DP there can be
an overlap of sub problems. By using memoization [maintaining a table of sub problems already solved),
dynamic programming reduces the exponential complexity to polynomial complexity $(O(n^2), O(n), etc.)$ for many
problems. The major components of DP are:

1. **Recursion:** Solves sub problems recursively.
2. **Memoization:** Stores already computed values in table (Memoization means caching).

> $$Dynamic Programming = Recursion + Memoization$$

Shortest Path in DAG

DAG: Stands for Directed Acyclic Graph which means, it has directed edges and contains no cycles.

> **Statement:** Let $G = (V, E)$ be a graph. Let $v_1, \ldots, v_n$ be an ordering of the vertices in V. $v_1, \ldots, v_n$ are in topologically sorted order if for all edges $(vi, vj) \in E, i < j$.

If G is a DAG, then we can find a topological sorting of the vertices. It is important to note that this ordering is not necessarily distinct. The shortest path to v_j is the shortest path to v_k, plus(+) the edge (v_k, v_j). We use this property to derive the required mathematical relations. We also know that $k < j$ since we have assumed that the vertices are in topologically sorted order.

Based on the above approach, we get the equation:

$$SP(v_i, v_j) = min_{v_k \in V}\{SP(v_i, v_k) + l(v_k, v_j)\}$$

Here, $l(v_k, v_j)$ is the length of the edge (v_k, v_j).

For each vertex v_j that is visited, we will potentially need to make a recursive call for each v_k where $k < j$ (this is the case if every vertex preceding v_j has an edge to v_j). Therefore, our recurrence relation is:

$$T(n) = \sum_{i<n} T(i) + O(n) \approx O(n^n)$$

This is calculated and computed to be slow as is a NP hard type equation. To fix this, we will solve the subproblems bottom-up instead of top-down. This will prevent us from needlessly solving the same subproblems multiple times, which is causing the slow runtime.

> **Algorithm:** Shortest path in DAG:
>
> function $SP(V, E, s)$
>
> $\{v_1, \ldots, v_n\} \leftarrow TOPSORT(V)$
>
> Assume: $v_i = s$
>
> $d[v_i] \leftarrow 0$
>
> for $v_j \neq v_i$ do
>
> $\quad d[v_j] \leftarrow \infty$
>
> for $j \leftarrow 1$ to n do
>
> $\quad$ for $k < j$ do
>
> $\quad\quad$ if $d[j] > d[k] + l(v_k, v_j)$ then
>
> $\quad\quad\quad d[j] \leftarrow d[k] + l(v_k, v_j)$

The result of this algorithm will be an array of values where each value is the shortest path in the DAG from s to the vertex corresponding to that index in the array. To calculate the value in location i, this algorithm takes $O(i)$ time. Summed over all locations in the array, the running time is $O(n^2)$.

Longest Increasing Subsequence(LIS)

> **Statement:** A subsequence of sequence $a_1, \ldots, a_n$ is some sequence $a_{i1}, \ldots, a_{ih}$ such that for all k, $1 \leq k \leq h$, we have $1 \leq i(h) \leq n$; and for any x_j in the subsequence, all a_i preceding a_j in the subsequence satisfy $i < j$. An increasing subsequence is a subsequence such that for any a_j in the subsequence, all a_i preceding a_j in the subsequence satisfy $a_i < a_j$. A largest increasing subsequence is a subsequence of maximum length.

In order to find the longest subsequence, let the input sequence be denoted as $v_1, \ldots, v_n$. We have the following two options:

1. v_n is in the subsequence.
2. v_n is NOT in the subsequence.
   2 is easy to account for, we just need to solve the same problem on a smaller sequence in order to ensure that we can recurse.

> To solve 1, we need to recurse on a slightly stronger problem: LIS(k) to be the longest increasing subsequence that ends at v_k. We get the following expression:

$LIS(k) = max_{j<k_{v_j<v_k}}\{LIS(j)\} + 1$

To solve the above, we find:
$LIS = max_k\{LIS(k)\}$

> Implementing this naively using recursion is slow. Instead, we use dynamic programming. We want to start with $k = 1$ and then increase k, instead of starting with $k = n$ and recursing.

> **Longest Increasing Subsequence:**
>
> **function** $LIS(v_1, \ldots, v_n)$
>
> $lis[1] \leftarrow 1$
>
> for $k \leftarrow 2$ to n do
>
> $lis[k] \leftarrow 0$
>
> if $lis[j] + 1 > lis[k]$ then
>
> $lis[k] \leftarrow lis[j] + 1$
>
> $lis \leftarrow 0$
>
> for $i \leftarrow 1$ to n do
>
> if $lis[k] > lis$ then
>
> $lis \leftarrow lis[k]$
>
> return lis

The runtime of this algorithm is $O(n^2)$.

Reason: The result of this algorithm will be an array of values where each value is the shortest path in the DAG from s to the vertex corresponding to that index in the array. To calculate the value in location i, this algorithm takes $O(i)$ time. Summed over all locations in the array, the running time is $O(n^2)$.
