The Floyd Warshall algorithm is a Dynamic Programming algorithm for the All Pairs Shortest Paths (APSP) Problem, specified below.

All Pairs Shortest Paths (APSP) Problem

Input. An edge-weighted directed graph G, with $|V(G)| = n$, $|E(G)| = m$ and a weight function $w : E(G) \to \mathbb{R}$.

Output. An $n \times n$ matrix D such that $D[u][v]$ contains the length of the shortest path from u to v, for all $u, v \in V(G)$. This matrix is called the *distance oracle*.

Recall that Dijkstra's algorithm and the Bellman-Ford algorithm only compute distances from a given source vertex to every other vertex in the graph. The problem solved by these algorithms is referred to as the Single Source Shortest Paths (SSSP) problem.

Instead, in the APSP problem, our objective is to compute the lengths of the shortest paths between *every pair* of vertices. These lengths are returned in a matrix called the *distance oracle*, where the term *oracle* implies that, given this matrix, we can find the distances between any *query* pair of vertices in *constant* time.

Alternative Outputs

Depending on the application, it may be desirable to further output the paths, instead of just the distances. The outputs corresponding to two such variants of the APSP problem are listed below and our algorithm for the APSP problem can easily be adapted to produce these outputs.

- **Routing Tables.** Consider another $n \times n$ matrix, say R, such that $R[u][v]$ contains the label of the *first vertex* along the shortest path from u to v. Using this matrix, we can easily plot the path $u \rightsquigarrow v$, for any $u, v \in V$, and as such, it *implicitly* stores the paths in the graph.
- **Shortest Path Trees.** Instead of a *global* matrix, we can instead store at node v the *shortest path tree* rooted at v. Indeed, this is no different than the shortest path tree obtained from Dijkstra's algorithm using v are the source node.

How to solve the APSP problem?

An obvious algorithm. For each vertex $v \in V(G)$, solve the SSSP problem using v as the source vertex. Using Dijkstra's algorithm, with a Fibonacci heap for finding the next closest vertex in each iteration, this gives an algorithm with complexity $\mathcal{O}\left(mn + n^2 \log n\right)$, while using the Bellman-Ford algorithm would give an algorithm with complexity $\mathcal{O}\left(mn^2\right)$.

As you will observe later, the Floyd Warshall algorithm instead has a complexity of $\mathcal{O}\left(n^3\right)$. As such, Dijkstra's algorithm would indeed outperform it as long as the input graph does not contain any negative weight edges. As with the Bellman-Ford algorithm, the Floyd Warshall algorithm can also work in the presence of negative weight edges, and can be used to detect the presence of negative weight cycles as well. But, if this is your objective, then using the Bellman-Ford algorithm would of course be faster.

Next, let us build towards the recursive formulation underlying the Floyd Warshall algorithm. We need to come up with a recursive function, say $\delta()$, such that $\delta(u, v, ?)$ would return the length of the shortest $u \rightsquigarrow v$ path. But, what should the extra argument/s '?' be?

Note. Understanding the final recursive formulation is actually easy, and you can safely skip to the end to understand the Floyd Warshall algorithm. But, doing so would not help improve your understanding of Dynamic Programming and will go against our objective.
My attempt here is to *mechanize* the process of applying the DP techniques such that, starting with a trivial inefficient recursive solution, we can gradually improve it to the most optimal algorithm.

First Attempt

A natural approach (ignoring efficiency for now) would be to have a third argument that contains an ordered sequence of vertices. Then, we can try and return the length of the path if the vertex sequence is a valid $u \rightsquigarrow v$ path. There are a couple of issues with this description.

- From a standard DP perspective, we would ideally want to start the recursion with the "longest" input. This won't work here as there need not even be such a long $u \rightsquigarrow v$ path. Instead, we would have to pick the shortest distance by going over all the possible vertex orderings.
- We are, in some sense, dealing with permutations of the vertices here. This would usually cause issues when trying to improve efficiency as permutations are inflexible and can only be recursed over in a specific order. This put restrictions on what can be done by the algorithm.

Interestingly, both Dijkstra's algorithm and the Bellman-Ford algorithm build upon this idea by considering just the *relevant* paths/permutations, which are few in number. In particular, the permutations can be effectively handled as the paths are *directed away from* the source vertex (this may be covered in more detail in one of the seminars).

Using a similar approach for the APSP problem would cause issues as we now have n different sources to direct our paths away from. Consequently, the reusability of precomputed distances could be very limited.

Second Attempt

As the problem with permutations is their rigid directed nature, it makes sense to relax our requirement and look at the elements in the paths without considering directions. In other words, we can try to have the *subsets* of V as the third parameter. In this case, what should the function return?

Note that, if we insist that the path be formed by every vertex in the current subset, this would again cause some of the same issues as in the previous case. Indeed, as before, there may not even exist a $u \rightsquigarrow v$ path that contains all the other vertices, which means $\delta(u, v, V)$ won't contain the length of the shortest $u \rightsquigarrow v$ path.

As such, we again need to relax our requirement. Considering that the details of the paths with few vertices should still be maintained while looking at the entire vertex set, it makes sense for

$\delta(u, v, S)$ to return the length of the shortest $u \rightsquigarrow v$ path that is formed by a *subset* of vertices in S, not necessarily the entire set S.

Try to think of a recursive formulation for $\delta()$ that works as per this specification.

- One possibility is to try and *guess* an element w that would lie on the shortest $u \rightsquigarrow v$ path that only contains vertices from S. Then, $\delta(u, w, S \setminus w)$ and $\delta(w, v, S \setminus w)$ would be the subproblems that need to be solved to obtain $\delta(u, w, S)$. And to *guess* w, we would need to try all possible elements w and pick the optimum one from them.
- In this recursive solution, do we need to ensure that the paths $u \rightsquigarrow w$ and $w \rightsquigarrow v$ do not share any intermediate vertices?
- To avoid this concern, we can consider an alternate recursive solution where we try to guess the *first* vertex w along the $u \rightsquigarrow v$ path, instead of any arbitrary vertex along the path. Indeed, corresponding to any vertex along the path, the shortest distance computed would be the same, and so, it is enough to pick just one such vertex, instead of computing for every vertex.

Observation. By adding this minor *greediness*, we are effectively reducing the branching factor of the recursion tree from an exponentially high value to something that is linear in the number of vertices. This was, of course, not our objective, but its an useful outcome nonetheless.

Observation 2. It looks like we are again building the path sequentially. Isn't this the same as the permutations case that was discussed earlier? It sort of is, but with the significant difference that the permutations are not the arguments and everything that is getting computed recursively is with respect to the subsets, not permutations.

So, is this recursive formulation suitable for applying the DP techniques to? Observe that, even though this formulation can be memoized, the DP table would have to have a size of $\Omega(n^2 2^n)$, and consequently, is not efficient.

Final Attempt

The issue in the previous attempt was that the function $\delta()$ depended on the 2^n subsets of V. But, we can still make a couple of useful observations to help us improve our recursive formulation.

Let X be the actual set of elements lying on the $u \rightsquigarrow v$ path. Then, for any set $S \subset V$,

- if $X \cap S \neq X$, then $\delta(u, v, S)$ will be greater than the length of the shortest $u \rightsquigarrow v$ path.
- if $X \subseteq S$, then $\delta(u, v, S)$ will be equal to the length of the shortest $u \rightsquigarrow v$ path.

These observations indicate that it might be enough to *incrementally* build the set S by adding vertices one by one. Formally, let $S_0, S_1, \ldots, S_n$ be subsets of V such that $S_0 = \emptyset$ and $S_n = V$, and each S_i, for $i \in [1, n)$, is obtained by adding a *new* vertex to S_{i-1}. (Note that the S_is can be formed by adding vertices in any arbitrary order. The sets would depend on the order, but our discussions would hold as long as they satisfy the given properties.)

Then, for each $u, v \in V$, it is sufficient to compute $\delta(u, v, S)$ where S is one of these $n + 1$ sets. If this is the case, as we shall discuss below, then the size of our DP table need only be $\mathcal{O}\left(n^3\right)$, not exponential.

Why is this enough?

Consider a pair of vertices $u, v \in V$ and a set $X \subseteq V \setminus \{u, v\}$ formed by the intermediate vertices along the shortest $u \rightsquigarrow v$ path. Let, S_i be the first set, out the $n+1$ sets described above, such that $X \subseteq S_i$. As per the observations made above, we have that

$$\delta(u, v, S_{<i}) > \delta(u, v, X) = \delta(u, v, S_i) = \delta(u, v, V).$$

And can we compute $\delta(u, v, S_i)$ recursively with respect to the $S_{<i}$ sets?
Let, $x = S_i \setminus S_{i-1}$. Also, let P^i_{uv} and P^{i-1}_{uv}, respectively, be the $u \rightsquigarrow v$ paths that only contain vertices from S_i and S_{i-1}.

How will P^i_{uv} and P^{i-1}_{uv} relate to each other? If P^i_{uv} *does not* contain the vertex x, then it is nothing but the shortest $u \rightsquigarrow v$ path that only contains vertices from $S_i \setminus x = S_{i-1}$; that is, it is nothing but the path P^{i-1}_{uv} itself. On the other hand, if x is an intermediate vertex in the path P^i_{uv}, then it must be formed by the sub-paths $u \rightsquigarrow x$ and $x \rightsquigarrow v$. Moreover, these paths would only contain vertices from S_{i-1}, as P^i_{uv} only contains vertices from $S_{i-1} \cup \{x\}$. Hence, in this case, $P^i_{uv} = P^{i-1}_{ux} \circ P^{i-1}_{xv}$, where $\circ$ is the path concatenation operator.

All these discussions lead us to our final recursive formulation, as given below.

Recursive Formulation

Base case.

$$\delta(u, v, 0) = \begin{cases} w(u, v) & \text{, if } (u, v) \in E(G) \\ \infty & \text{, otherwise} \end{cases}$$

If $i > 0$, then

$$\delta(u, v, i) = \min \begin{cases} \delta(u, v, i-1) \\ \delta(u, v_i, i-1) + \delta(v_i, v, i-1) \end{cases}$$

where, v_i is the i^{th} vertex in $V(G)$.

Note that, instead of directly dealing with sets as in the preceding discussions, we are arbitrarily labeling the vertices from 1 to n and using the argument i for $\delta()$ to denote the set S_i.

Finally, applying our DP techniques to this recursive formulation will result in the standard version of the Floyd Warshall algorithm. These will be set as a homework for you to complete.

General Note. When trying to recurse over a subset of elements, iteratively adding the elements one by one, as was done here, is a fairly common approach seen in DP algorithms. We will again design a similar recursive formulation for the 0-1 Knapsack Problem as well.

The Floyd Warshall Algorithm

The standard version of the Floyd Warshall algorithm is given below for reference. As mentioned above, you can apply the DP techniques to our recursive formulation to obtain this algorithm.

```
$\underline{FloydWarshall(G(V,E))}$
*Initialization:* Maintain a matrix $D[][]$ of size $|V| \times |V|$.
1: $D[u][v] = \begin{cases} 0 & ,\text{if } u = v \\ w(u,v) & ,\text{if } (u,v) \in E(G) \\ \infty & ,\text{otherwise} \end{cases}$
2: for $i = 1$ to $n$ do
3:     for $u = 1$ to $n$ do
4:         for $v = 1$ to $n$ do
5:             $D[u][v] = \min \begin{cases} D[u][v] \\ D[u][i] + D[i][v] \end{cases}$
6:         end for
7:     end for
8: end for
9: return $D$
```

Alternate View of the Algorithm

Thanks to the bottom up algorithm given above, we can think of the algorithm as starting with an empty graph on the vertex set V, with all the distances initialized to ∞. Then, in the i^{th} iterative step, we add all the edges incident to, and from, the i^{th} vertex to the graph and update the distances, as required.

(This is very similar in idea to another class of algorithms known as dynamic algorithms.)