Dynamic Programming Solution with Pseudo-code

Weighted Job Scheduling is a problem where you are given a set of presentations with start and end times, and a weight (here, popularity score) associated with each presentation. The goal is to find the maximum popularity score that can be earned by scheduling presentations such that no two presentations overlap.

1. **Sort the presentations by their finish time**: First, sort the presentations in non-decreasing order of their finish times. This will make it easier to find the next presentation that can be scheduled after the current presentation.

2. **Find the latest non-conflicting presentation**: For each presentation, find the latest presentation that does not conflict with it. This can be done by iterating through the sorted list of presentations in reverse order and checking if the start time of the current presentation is greater than or equal to the finish time of the previous presentation.

3. **Create a table to store the maximum profit**: Create a table `dp` of size `n`, where `n` is the number of presentations, to store the maximum popularity score that can be earned by scheduling presentations from 0 to `i`.
   Initialize `dp[0]` to the popularity score of the first presentation.

4. **Fill the table using dynamic programming**: For each presentation `i` from 1 to `n-1`, calculate the maximum popularity score that can be achieved by either including or excluding the current presentation. If the current presentation is included, then add its weight to the maximum profit earned by scheduling presentations up to the latest non-conflicting presentation. If the current presentation is excluded, then the maximum profit remains unchanged.
   Store the maximum of these two values in `dp[i]`.
5. **Return the maximum popularity score**: The maximum popularity score that can be earned by scheduling presentations is stored in `dp[n-1]`.

Pseudo-code

```
weighted_presentation_scheduling(presentations):
    # Sort presentations by finish time
    sort presentations by finish time

    # Find latest non-conflicting presentation for each presentation
    for each presentation i:
        for each presentation j before i in reverse order:
            if start time of presentation j <= finish time of presentation i:
                store index of presentation j as latest non-conflicting presentation for presentation i
                break

    # Create table to store maximum popularity score
    dp[0] = weight of first presentation

    # Fill table using dynamic programming
    for each presentation i from 1 to n-1:
        include = weight of presentation i
        if latest non-conflicting presentation for presentation i exists:
            include += dp[latest non-conflicting presentation for presentation i]
        exclude = dp[i-1]
        dp[i] = max(include, exclude)

    # Return maximum popularity score
    return dp[n-1]
```