# 2830. Maximize the Profit as the Salesman

Medium Array Binary Search Dynamic Programming Sorting

Leetcode Link

## Problem Description

In this problem, you have an area consisting of a number of houses in a row, specifically `n` houses, indexed from `0` to `n - 1`. There's an opportunity to sell these houses to various buyers. Buyers come with various offers, and each buyer is represented by three components in an array: the `start` index, the `end` index, and the `gold` they are willing to pay. The `start` and `end` indices indicate the range of houses that the buyer is interested in purchasing, inclusive. The goal is to maximize the amount of gold you earn by selling these houses to the buyers.

However, there are two constraints to consider:

1. A house can only be sold to one buyer.
2. Some houses may not be sold at all if that maximizes the total earnings.

You need to formulate a strategy that selects which offers to accept to maximize your profit and return the maximum amount of gold you can earn.

## Intuition

The intuition behind the solution involves solving the problem using *dynamic programming*. We're looking to maximize earnings, but we also need to ensure that the same house isn't sold to different buyers. This problem appears similar to the "Weighted Interval Scheduling" problem, where we aim to find the maximum weight (in this case, gold) of non-overlapping intervals (here, the house ranges).

Here is the general approach to arrive at the solution:

1. Sort all the offers by their `end` index. This allows us to process the offers in order and use a greedy approach to consider the latest offers without looking back to previous offers that end later.
2. Initialize an array `f` to keep track of the maximum gold that can be accumulated by considering offers up to the current index. This is the *dynamic programming* table, where `f[i]` represents the maximum earnings by considering offers up to the `i-th` offer.
3. Iterate over the sorted offers, and for each offer, find the previous non-conflicting offer's index `j` using binary search (`bisect_left` in Python), which will give the index of the rightmost offer that doesn't overlap with the current one.
4. Update the dynamic programming table `f` for the current offer index `i` by making a decision: either take the current offer added to the best we could do up to offer `j`, or stick with the best earnings up to the previous offer `i-1`. This is expressed as:

```
f[i] = max(f[i - 1], f[j] + v)
```

   Here, `v` is the amount of gold of the current offer. This decision ensures we either extend our current earnings with the current offer, if it's beneficial, or we do not sell to the current buyer if it doesn't increase our earnings.
5. The maximum gold earned is then the last value in the `f` array after considering all offers.

By sorting the offers, using *dynamic programming* to store intermediate results, and binary search to find non-overlapping intervals efficiently, we are able to find the solution in a time-complex manner, which significantly reduces the time complexity compared to a naive approach that might check every possible combination of sales.

## Solution Approach

To solve this problem using the described intuition, let's walk through the implementation details step by step:

1. **Sorting Offers**: The first step involves sorting the `offers` array based on the `end` index of each offer. We do this because once we've processed an offer, we don't have to go back and consider earlier offers that may overlap with newer offers. We use the `sort()` method of a list in Python with a lambda function as the `key` to sort the offers by their `end` index.

```
offers.sort(key=lambda x: x[1])
```

2. **Dynamic Programming Table Initialization**: We initialize a dynamic programming table `f` with a length of `len(offers) + 1`. The first element is implicitly `0`, as no gold can be earned without accepting any offers. We also create a list `g` which contains just the `end` indices of the sorted offers for quicker access during binary search.

```
f = [0] * (len(offers) + 1)
g = [x[0] for x, _, _ in offers]
```

3. **Iterating Through Offers**: We iterate through the offers using an `enumerate` function starting from `1`, since our dynamic programming table starts from index `1`. This iteration gives us each offer's start index, end index, and gold alongside the iteration index `i`.

```
for i, (x, _, v) in enumerate(offers, 1):
```

4. **Binary Search for Non-Conflicting Offers**: Within the loop, for each offer, we use the `bisect_left` function to perform a binary search on list `g`. We're looking for the rightmost offer that does not conflict with the current offer based on its starting index. This efficiently finds us the index `j` for the previous offer that doesn't overlap with the current one.

```
j = bisect_left(g, x)
```

5. **Dynamic Programming Decision**: The core of the solution is the dynamic programming decision made in each iteration. For each offer, we decide if adding its value to the best value we could obtain before it (cumulative profit up to offer `j`) is better than the maximum gold we can achieve without it (`f[i - 1]`).

```
f[i] = max(f[i - 1], f[j] + v)
```

6. **Returning the Final Result**: After the loop, the last element in the `f` list represents the maximum gold we can earn by selling the houses according to the problem constraints. This value is returned as the final result.

```
return f[-1]
```

In terms of algorithms and patterns, the solution involves a dynamic programming approach to track and maximize profits, a greedy approach to prioritize processing later offers, and a binary search for efficiently finding compatible offers. Data structures used include lists (arrays in some other languages) for sorting offers and keeping track of dynamic states in the decision process.

### Example Walkthrough

Let's consider an example to illustrate the solution approach. Assume we have the following offers for 4 houses indexed from 0 to 3:

```
offers = [[0, 1, 10], [0, 2, 15], [2, 3, 18], [1, 3, 20]]
```

Here's how we would walk through the solution:

1. **Sorting Offers**: We sort the `offers` array by the `end` index:

```
offers after sorting = [[0, 1, 10], [0, 2, 15], [1, 3, 20], [2, 3, 18]]
```

2. **Dynamic Programming Table Initialization**: We initialize our dynamic programming table `f` with zeroes and an additional element for the base case, `f = [0, 0, 0, 0, 0]`. We also create a list `g` of end indices `[1, 2, 3, 3]`.

3. **Iterating Through Offers**: We start iterating through the sorted offers:

   - For the first offer `[0, 1, 10]`, there are no previous non-conflicting offers, so `f[1] = max(f[0], 0 + 10) = 10`.
   - For the second offer `[0, 2, 15]`, similarly, there are no previous non-conflicting offers, so `f[2] = max(f[1], 0 + 15) = 15` (since 15 is greater than 10).
   - For the third offer `[1, 3, 20]`, we use binary search to find the index of non-conflicting offer. The search returns 2, which means the first offer is not conflicting. So, `f[3] = max(f[2], f[1] + 20) = max(15, 10 + 20) = 30`.
   - For the fourth offer `[2, 3, 18]`, the search returns the same index since the previous offer is overlapping, we cannot add the current offer to it. So, `f[4] = max(f[3], f[2] + 18) = max(30, 15 + 18) = 30`.

4. **Returning the Final Result**: After iterating through all offers, the last value in `f` is `30`, which is the maximum gold we can earn.

This walkthrough shows how dynamic programming is used to keep track of maximum earnings at each step by either taking the new offer or sticking with the previous best option. The binary search is key to quickly finding the best prior non-conflicting offer that we can use as a base for adding the new offer's gold.

## Python Solution

```
from bisect import bisect_left
from typing import List

class Solution:
    def maximize_the_profit(self, n: int, offers: List[List[int]]) -> int:
        # Sort the offers based on their ending times
        offers.sort(key=lambda offer: offer[1])

        # Initialize an array to store the maximum profit
        # that can be obtained up to each offer index
        max_profits = [0] * (len(offers) + 1)

        # Extract the ending times to facilitate binary search later
        ending_times = [offer[1] for offer in offers]

        # Iterate through each offer
        for i, (start, end, profit_value) in enumerate(offers, 1):
            # Find the index of the latest offer that ends before the current offer starts
            index = bisect_left(ending_times, start_time)

            # Update the maximum profit for the current position
            # It's either we take the current offer and its value to the profit
            # at the aforementioned index, or we ignore the current offer and carry
            # forward the previous maximum profit.
            max_profits[i] = max(max_profits[i - 1], max_profits[index] + value)

        # Return the maximum profit at the end of the array.
        # which is the profit after considering all offers.
        return max_profits[-1]
```

## Java Solution

```
class Solution {

    // Method that aims to maximize the profit based on the offers list and return the maximum profit.
    public int maximizeTheProfit(int n, List<List<Integer>> offers) {
        // Sort the offers based on their end times.
        offers.sort((a, b) -> a.get(1) - b.get(1));

        // Initialize a dynamic programming array to store the maximum profit up to each offer.
        int[] maxProfitUpTo = new int[offers.size() + 1];

        // Create an array to store the end times of each offer.
        int[] endTimes = new int[offers.size()];
        for (int i = 0; i < offers.size(); ++i) {
            endTimes[i] = offers.get(i).get(1);
        }

        // Calculate the maximum profit for each offer.
        for (int i = 1; i <= offers.size(); ++i) {
            var currentOffer = offers.get(i - 1);
            // Find the latest offer that does not conflict with the current offer.
            int latestNonConflictingIndex = binarySearchEndTimes(endTimes, currentOffer.get(0));
            // Update the maximum profit for the current offer, considering not taking or taking the current offer.
            maxProfitUpTo[i] = Math.max(maxProfitUpTo[i - 1], maxProfitUpTo[latestNonConflictingIndex] + currentOffer.get(2));
        }

        // Return the maximum profit after considering all offers.
        return maxProfitUpTo[offers.size()];
    }

    // Binary search method to find the minimum index of offers that end before the given start time x.
    private int binarySearchEndTimes(int[] times, int startTime) {
        int left = 0, right = times.length;
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to dividing by 2.
            if (times[mid] >= startTime) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

## C++ Solution

```
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to maximize the profit based on the given offers.
    int maximizeTheProfit(int totalOffers, std::vector<std::vector<int>>& offers) {
        // Sort the offers based on the ending time
        std::sort(offers.begin(), offers.end(), [](const std::vector<int>& a, const std::vector<int>& b) {
            // Compare the second element of each vector, which is the ending time
            return a[1] < b[1];
        });

        // Update the totalOffers to the size of the offers vector
        totalOffers = offers.size();

        // Dynamic programming array to store the maximum profit at each step
        std::vector<int> maxProfitInEachStep(totalOffers + 1, 0);

        // Vector to store the end times of the offers
        std::vector<int> endTimes;
        for (const auto& offer : offers) {
            // Populate the endTimes vector with the end times of the offers
            endTimes.push_back(offer[1]);
        }

        // Loop through each offer to calculate the maximum profit
        for (int i = 1; i <= totalOffers; ++i) {
            // Current offer details: start time, end time, and profit
            auto currentOffer = offers[i - 1];
            // Find the last offer that finishes before the current one starts
            int lastNonConflictingOfferIndex = std::lower_bound(endTimes.begin(), endTimes.end(), currentOffer[0]) - endTimes.begin();
            // Update the maxProfit at i considering the current offer
            maxProfit[i] = std::max(maxProfit[i - 1], maxProfit[previousAvailable] + currentOffer[2]);
        }

        // The last element of maxProfit contains the maximum profit possible
        return maxProfitInEachStep[totalOffers];
    }
};
```

## Typescript Solution

```
function maximizeTheProfit(totalOffers: number, priceOffers: number[][]): number {
    // Sort the offers by the second element of each offer tuple, which is the sell time.
    priceOffers.sort((offerA, offerB) => offerA[1] - offerB[1]);

    // The total number of offers is updated to the length of the provided offer list.
    totalOffers = priceOffers.length;

    // Initialize an array for dynamic programming to store the maximum profit till each offer.
    const maxProfitTillOffer: number[] = Array(totalOffers + 1).fill(0);

    // An array to store the sell time of each offer. This is used for binary search.
    const sellTimes = priceOffers.map(offer => offer[1]);

    // Helper function to perform binary search for the index of the first sell time that is >= x.
    const findSellTimeIndex = (targetTime: number): number => {
        let left = 0,
            right = totalOffers;
        while (left < right) {
            let mid = (left + right) >> 1; // Same as Math.floor((left + right) / 2)
            if (sellTimes[mid] >= targetTime) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Calculate the maximum profit for each offer using dynamic programming.
    for (let i = 1; i <= totalOffers; ++i) {
        // Using binary search to find the latest offer which sell time is less than or equal to the current offer's buy time.
        const [, buyTime, profit] = priceOffers[i - 1];
        const previousAvailableOfferIndex = findSellTimeIndex(buyTime);

        // Update the maximum profit at this index by comparing:
        // - The maximum profit till the previous offer, and
        // - The maximum profit till the available offer plus the current offer's profit.
        maxProfitTillOffer[i] = Math.max(maxProfitTillOffer[i - 1], maxProfitTillOffer[previousAvailableOfferIndex] + priceOffers[i - 1][2]);
    }

    // The last element in maxProfitTillOffer contains the maximum profit that can be achieved.
    return maxProfitTillOffer[totalOffers];
}
```

## Time and Space Complexity

### Time Complexity

The given code has several different operations, each with its own time complexity, which we should analyze step by step:

1. Sorting the offers: `offers.sort(key=lambda x: x[1])` sorts the list based on the second element of the sublists. The sorting algorithm used by Python's `sort()` function is Timsort, which has a time complexity of $O(n \log n)$, where $n$ is the length of the list to sort. Since all offers are being sorted, this step has a time complexity of $O(n \log n)$.
2. Preparing a list for dynamic programming: `f = [0] * (len(offers) + 1)` initializes a list with `len(offers) + 1` elements, each being `0`. This has a time complexity of $O(n)$, where $n$ is the length of the offers list plus one. However, this is overshadowed by the sorting step, as list initialization is linear $O(n)$ and $m$ would be equal to $n$ for the purpose of comparison.
3. Creating another list for binary search: `g = [x[0] for x in offers]` creates a list of end times for binary search. This step has a time complexity of $O(n)$ as it involves going through each of the $n$ offers.
4. Iterating over the offers and applying dynamic programming with a binary search: The loop `for i, (x, _, v) in enumerate(offers, 1):` iterates $n$ times. Inside this loop, `j = bisect_left(g, x)` performs a binary search which has a logarithmic time complexity, $O(\log n)$, and the dynamic programming state update `f[i] = max(f[i - 1], f[j] + v)` is performed in $O(1)$ time. Hence, the nested operation's time complexity is $O(n \log n)$ because the binary search is called per iteration over the $n$ elements.

Total time complexity is dominated by the sorting and the loop with binary search, both $O(n \log n)$, so the overall time complexity is $O(n \log n)$.

### Space Complexity

The space complexity is determined by the space used to store data structures in the algorithm:

1. The sorted offers list does not require additional space as it is sorted in-place.
2. The dynamic programming list `f` of size `len(offers) + 1` adds $O(n)$ space complexity.
3. The list `g`, used for binary search which retains the end times of offers, also adds another $O(n)$ space complexity.
4. The stack space used in sorting and binary search is $O(\log n)$, due to the recursive nature of these algorithms (assuming the worst-case scenario), but this is generally considered a lesser term compared to the previous space allocations.

Total space complexity is the sum of the space used, which is $O(n)$ with respect to the number of offers ($n$ being the length of the offers list).

In conclusion, the overall time complexity is $O(n \log n)$ and space complexity is $O(n)$.