2830. Maximize the Profit as the Salesman Medium (Array) Binary Search Dynamic Programming Sorting

Leetcode Link

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

Problem Description

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:

2. Some houses may not be sold at all if that maximizes the total earnings.

A house can only be sold to one buyer.

- 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.

naive approach that might check every possible combination of sales.

- 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 1-1. This is expressed as: 1 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

Solution Approach

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

sort() method of a list in Python with a lambda function as the key to sort the offers by their end index.

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

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

programming table starts from index 1. This iteration gives us each offer's start index, end index, and gold alongside the iteration

end indices of the sorted offers for quicker access during binary search. f = [0] * (len(offers) + 1)

index i.

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

2 g = [x[1] for x in offers]

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

maximum gold we can achieve without it (f[i - 1]).

3. Iterating Through Offers: We iterate through the offers using an enumerate function starting from 1, since our dynamic

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. 1 j = bisect_left(g, s)

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

6. Returning the Final Result: After the loop, the last element in the flist represents the maximum gold we can earn by selling the

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.

houses according to the problem constraints. This value is returned as the final result. 1 return f[-1]

Example Walkthrough

Here's how we would walk through the solution:

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

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

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

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

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

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

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].

 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

• For the fourth offer [2, 3, 10], the search returns the same index since the previous offer is overlapping, we cannot add

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

For the first offer [0, 1, 10], there are no previous non-conflicting offers, so f[1] = max(f[0], 0 + 10) = 10.

means the first offer is not conflicting. So, f[3] = max(f[2], f[1] + 20) = max(15, 10 + 20) = 30.

the current offer to it. So, $f[4] = \max(f[3], f[2] + 10) = \max(30, 15 + 10) = 30$.

that we can use as a base for adding the new offer's gold.

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

 $maximum_profits = [0] * (len(offers) + 1)$

Iterate through each offer

return maximum_profits[-1]

Initialize an array to store the maximum profit

Extract the ending times to facilitate binary search later

for i, (start_time, end_time, value) in enumerate(offers, 1):

Update the maximum profit for the current position

index = bisect_left(ending_times, start_time)

Return the maximum profit at the end of the array,

public int maximizeTheProfit(int n, List<List<Integer>> offers) {

// Create an array to store the end times of each offer.

// Return the maximum profit after considering all offers.

// Sort the offers based on their end times.

int[] maxProfitUpTo = new int[offers.size() + 1];

offers.sort((a, b) -> a.get(1) - b.get(1));

int[] endTimes = new int[offers.size()];

return maxProfitUpTo[offers.size()];

// Vector to store the end times of the offers

// Add the end time of each offer to the endTimes vector

// Current offer details: start time, end time, and profit

// Update the maxProfit at i considering the current offer

// The last element of maxProfit contains the maximum profit possible

function maximizeTheProfit(totalOffers: number, priceOffers: number[][]): number {

priceOffers.sort((offerA, offerB) => offerA[1] - offerB[1]);

const sellTimes = priceOffers.map(offer => offer[1]);

if (sellTimes[mid] >= targetTime) {

const maxProfitAtIndex: number[] = Array(totalOffers + 1).fill(0);

// Sort the offers by the second element of each offer tuple, which is the sell time.

// Initialize an array for dynamic programming to store the maximum profit till each offer.

// The total number of offers is updated to the length of the provided offer list.

// An array to store the sell time of each offer. This is used for binary search.

// Calculate the maximum profit for each offer using dynamic programming.

// Update the maximum profit at this index by comparing:

// - The maximum profit till the previous offer, and

const latestCompatibleOfferIndex = findSellTimeIndex(priceOffers[i - 1][0]);

// The last element in maxProfitAtIndex contains the maximum profit that can be achieved.

list to sort. Since all offers are being sorted, this step has a time complexity of $O(n \log n)$.

// - The maximum profit till the latest compatible offer plus the current offer's profit.

// Find the last offer that finishes before the current one starts

maxProfit[i] = std::max(maxProfit[i - 1], maxProfit[prevAvailable] + currentOffer[2]);

// Loop through each offer to calculate the maximum profit

std::vector<int> endTimes;

for (auto& offer : offers) {

return maxProfit[totalOffers];

totalOffers = priceOffers.length;

right = mid;

left = mid + 1;

for (let i = 1; i <= totalOffers; ++i) {</pre>

return maxProfitAtIndex[totalOffers];

Time and Space Complexity

} else {

return left;

endTimes.push_back(offer[1]);

for (int i = 1; i <= totalOffers; ++i) {</pre>

auto currentOffer = offers[i - 1];

for (int i = 0; i < offers.size(); ++i) {</pre>

endTimes[i] = offers.get(i).get(1);

which is the profit after considering all offers

Find the index of the latest offer that ends before the current offer starts

maximum_profits[i] = max(maximum_profits[i - 1], maximum_profits[index] + value)

// Method that aims to maximize the profit based on the offers list and return the maximum profit.

// Initialize a dynamic programming array to store the maximum profit up to each offer.

int latestNonConflictingIndex = binarySearch(endTimes, currentOffer.get(0));

It's either we take the current offer and add its value to the profit

at the aforementioned index, or we ignore the current offer and carry

that can be obtained up to each offer index

ending_times = [offer[1] for offer in offers]

forward the previous maximum profit

Python Solution

10

11

12

13

14

15

16

17

18

20

21

22

23

24

25

26

27

28

30

8

9

10

12

13

14

15

22

23

24

25

26

27

28

29

19

20

21

22

23

25

26

27

28

29

30

32

33

34

35

36

37

38

39

41

8

9

10

11

12

13

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

43

42 }

40 };

- 1 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
- Java Solution class Solution {

// 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));

```
16
           // Calculate the maximum profit for each offer.
            for (int i = 1; i <= offers.size(); ++i) {</pre>
18
                var currentOffer = offers.get(i - 1);
19
20
                // Find the latest offer that does not conflict with the current offer.
21
```

```
30
31
32
       // Binary search method to find the maximum index of offers that end before the given start time x.
33
       private int binarySearch(int[] times, int startTime) {
34
           int left = 0, right = times.length;
35
           while (left < right) {</pre>
               int mid = (left + right) >> 1; // Equivalent to dividing by 2.
36
37
               if (times[mid] >= startTime) {
38
                    right = mid;
               } else {
39
                   left = mid + 1;
40
41
           return left;
43
44
45 }
46
C++ Solution
 1 #include <vector>
 2 #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) {
 9
               // If a finishes before b, then a should come before b
10
               return a[1] < b[1];
11
           });
12
13
           // Update the totalOffers to the size of the offers vector
14
15
           totalOffers = offers.size();
16
           // Dynamic programming array to store the maximum profit at each step
17
           std::vector<int> maxProfit(totalOffers + 1, 0);
```

int prevAvailable = std::lower_bound(endTimes.begin(), endTimes.end(), currentOffer[0]) - endTimes.begin();

// Using binary search to find the latest offer which sell time is less than or equal to the current offer's buy time.

maxProfitAtIndex[i] = Math.max(maxProfitAtIndex[i - 1], maxProfitAtIndex[latestCompatibleOfferIndex] + priceOffers[i - 1][2])

// Helper function to perform binary search for the index of the first sell time that is >= x. const findSellTimeIndex = (targetTime: number): number => { 15 16 let left = 0; let right = totalOffers; 17 while (left < right) {</pre> const mid = (left + right) >> 1; // Same as Math.floor((left + right) / 2)

};

Typescript Solution

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

Time Complexity

3. Creating another list for binary search: g = [x[1] 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, (s, _, v) in

the sorting step, as list initialization is linear O(m) and m would be equal to n for the purpose of comparison.

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.

enumerate(offers, 1): iterates n times. Inside this loop, $j = bisect_left(g, s)$ performs a binary search which has a

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

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(m), where m is the length of the offers list plus one. However, this is overshadowed by

- 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**
- 1. The sorted offers list does not require additional space as it is sorted in-place.

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

- 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.

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

2. The dynamic programming list f of size len(offers) + 1 adds O(n) space complexity.

offers list).

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