983. Minimum Cost For Tickets

Medium Array Dynamic Programming Leetcode Link

Problem Description

In this train traveling problem, you are given an integer array `days`, which represents each day in the year you will travel by train. The days are within the range from 1 to 365. The goal is to find the minimum cost needed to purchase train tickets to cover all those traveling days within a year.

There are three types of train passes you can buy:

- A 1-day pass for `costs[0]` dollars.
- A 7-day pass for `costs[1]` dollars.
- A 30-day pass for `costs[2]` dollars.

A pass allows you to travel for that number of consecutive days. For instance, if you buy a 7-day pass on day 2, you can travel from day 2 to day 8 inclusively.

You need to determine the minimum total cost to buy passes such that you can travel on all the days listed in the array `days`.

Intuition

To tackle this problem, we need to consider that buying a longer-term pass (7-day or 30-day) might be cheaper in the long run even if we don't use all of its valid days. Our solution must weigh the costs and benefits of each type of pass for each travel day.

We can think about the problem in terms of decision-making on each traveling day: should we buy a 1-day, 7-day, or 30-day pass? After buying a pass, we'd like to know "when do we have to make the next decision?" This leads to a recursive solution where we try all possible pass purchases and choose the one with the lowest overall cost.

For each day in `days` we are traveling, we have three choices: buy a 1-day pass, buy a 7-day pass, or buy a 30-day pass. After we buy a pass, we will skip to the next day that is outside the range of the pass. For example, if we're considering buying a 7-day pass on day `i`, the next decision would be made for the first day after those 7 days.

The `mincostTickets` function uses dynamic programming to avoid recalculating the minimum cost for days that have already been processed. A `dfs` function is used to find the minimum cost recursively, starting from the first day in `days`. The `@cache` decorator is used to memoize results, which saves the results of subproblems and dramatically reduces the time complexity.*

An array could also be used to cache results, but using the `@cache` decorator is cleaner and avoids the need to explicitly manage the caching logic.

Using binary search (`bisect_left` from the `bisect` module), we determine the day to continue the search from after buying a particular pass. The `bisect_left` function finds the position to insert an element to keep the list sorted and is used here to find the first travel day that is not covered by the current pass being considered.

The recursion's base case is when `i` goes beyond the last travel day, returning a cost of `0` because no further tickets need to be purchased.

We iterate through all types of passes, calculate the total cost if we bought a particular type of pass, and then take the minimum of those costs. The result from the `dfs` function called with 0, the first day, gives us the minimum cost required for all the travel days.

*As of the knowledge cutoff date in September 2021, the `@cache` decorator is available in Python 3.9 and later as part of the `functools` module.

Solution Approach

The solution takes a dynamic programming approach using recursion with memoization to keep track of the subproblems that have already been solved. Here's a breakdown of the implementation:

1. **Recursion**: The core algorithm is a recursive function `dfs(i)`, which computes the minimum cost starting from the `i`-th travel day in the `days` array. If `i` is equal to or greater than the length of `days`, it means all travel days have been covered, and the function returns `0`.

2. **Memoization**: The use of the `@cache` decorator from the `functools` module automatically stores the results of the recursive calls. This way, when the function is called again with the same `i`, it returns the stored result instead of recomputing it. This memoization significantly speeds up the execution by eliminating duplicate calculations.

3. **Binary Search**: The `bisect_left` function from the `bisect` module efficiently finds the smallest index `j` at which the day `days[i] + d` could be inserted to maintain the sorted order. Here, `d` represents the duration of the pass (1, 7, or 30 days), and finding `j` allows us to determine the first day that is not covered by the pass. This helps us move to the next subproblem without iterating over each day.

4. **Minimum Cost Calculation**: The recursive function tries to simulate buying each type of pass (1-day, 7-day, or 30-day), and for each case, it adds the cost of the pass `c` to the result of the recursive call for the next uncovered day `dfs(j)`. Then, it stores the minimum of these calculated costs in the variable `res`.

The dynamics of the function are as follows:

- We iterate over each type of pass using a loop `for c, d in zip(costs, [1, 7, 30]):`.
- We find the next index `j` to call our function recursively via `j = bisect_left(days, days[i] + d)`.
- We calculate the cost of buying the current pass and add it to the cost of all future decisions: `res = min(res, c + dfs(j))`.
- After trying all pass options for day `i`, `res` will contain the minimum cost to cover day `i` and all subsequent days.

After the setup of the recursive function and decorators, the solution is initiated by calling `return dfs(0)`. The result is the minimum total cost to purchase train tickets for all the traveling days.

This algorithm efficiently decides the best pass to buy for each travel day, considering all the possible options and their costs, to come up with the minimum total cost for all travel days.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume the `days` array in which we will travel is given by `[1, 4, 6, 7, 8, 20]`, and the costs of train passes are available as `[2, 7, 15]` for 1-day, 7-day, and 30-day passes, respectively.

We start by calling the `dfs` function with `i = 0`, which corresponds to the first traveling day, day `1`.

1. **Day 1 Decision:**
   - Buy a 1-day pass for `costs[0]` or `$2`. Our next decision will be on day `4` (next travel day after day `1`).
   - Buy a 7-day pass for `costs[1]` or `$7`. The next decision day will be day `20` (as days `4`, `6`, `7`, and `8` all fall within the 7-day period starting day `1`).
   - Buy a 30-day pass for `costs[2]` or `$15`. Since the 30-day pass covers the entire `days` array in this example, this would mean no more decisions are necessary.
2. **Subsequent Decisions:**
   - If we bought a 1-day pass for the first day, on day `4`, we face similar options (buy a 1-day, 7-day, or 30-day pass), and the same applies for days `6`, `7`, `8`, and `20`. Each option's cost is calculated and added to the initial `$2`.
   - If we chose a 7-day pass, we next buy a pass on day `20`. Here, since it's the last day, we'd only consider a 1-day pass, which adds `$2` to the initial `$7`.
3. **Memoization:**
   - During the process, all calculated costs are stored, and if a day is reached that has been previously calculated, we simply use the stored value.
4. **Recursive Calls and Minimum Cost Calculation:**
   - The `dfs(4)` call calculates the minimum cost from day `4` and so on. For example, if `dfs(4)` returns `$9` (cheapest way to cover days `4, 6, 7, 8, 20`), the total cost of buying a 1-day pass on day `1` and using the optimal strategy onward is `$2 + $9 = $11`.
5. **Final Decision:**
   - After exploring all options, we take the minimum total cost to purchase train tickets for all travel days. In our case, it's min(`$2` + `dfs(4)`, `$7` + `dfs(20)`, `$15`).

Assuming `dfs(4)` returns `$9` (example value), `dfs(20)` returns `$2`, the final costs would be `$11`, `$9`, and `$15`, and the minimum cost is thus `$9`. So buying a 7-day pass on day `1` and a 1-day pass on day `20` would be the optimal solution here.

Python Solution

```python
from bisect import bisect_left
from functools import lru_cache

class Solution:
    def mincostTickets(self, days: List[int], costs: List[int]) -> int:
        # Using lru_cache from functools to memoize results of recursive calls
        # This will store results of the subproblems so they do not need to be recomputed
        @lru_cache(maxsize=None)
        def min_cost_from_day(index):
            # Base case: when all days have been covered
            if index >= len(days):
                return 0

            # Initialize result as infinity to ensure any minimum will be taken
            result = float('inf')

            # Iterate over each type of ticket to cover future travel
            for ticket_cost, validity_duration in zip(costs, [1, 7, 30]):
                # Find the next day index when the current ticket will be expired
                next_index = bisect_left(days, days[index] + validity_duration)
                # Calculate the cost if we select current ticket and call recursively for the remaining days
                total_cost = ticket_cost + min_cost_from_day(next_index)
                # Update the minimum result
                result = min(result, total_cost)

            # Return the minimum cost found
            return result

        # Start from the first day we have in the list
        return min_cost_from_day(0)

```

Java Solution

```java
public class Solution {

    // Define constants for pass durations
    private static final int[] PASS_DURATIONS = new int[]{1, 7, 30};

    // Variables to store costs, days, and memoized values
    private int[] passCosts;
    private int[] travelDays;
    private int[] memo;
    private int totalDays;

    // Method to calculate the minimum cost of tickets for given travel days and ticket costs
    public int mincostTickets(int[] days, int[] costs) {
        totalDays = days.length;
        memo = new int[totalDays];
        this.passCosts = costs;
        this.travelDays = days;

        // Fill memo array with default values to denote not calculated
        Arrays.fill(memo, -1);

        // Start DFS from the first day
        return dfs(0);
    }

    // Helper method to perform Depth-First Search (DFS)
    private int dfs(int currentIndex) {
        // Base case: if the currentIndex is beyond the last day, no cost is needed
        if (currentIndex >= totalDays) {
            return 0;
        }

        // If the cost has already been computed, return it
        if (memo[currentIndex] != -1) {
            return memo[currentIndex];
        }

        // Initialize result as the maximum value
        int result = Integer.MAX_VALUE;

        // Consider all types of passes
        for (int k = 0; k < 3; ++k) {
            // Find the index of the next day right after the pass expires
            int nextIndex = lowerBound(travelDays, travelDays[currentIndex] + PASS_DURATIONS[k]);

            // Calculate the minimum cost using the chosen pass
            result = Math.min(result, passCosts[k] + dfs(nextIndex));
        }

        // Save result to memo array
        memo[currentIndex] = result;

        // Return the final minimized cost
        return result;
    }

    // Helper method to find the lower bound index for a given day (binary search)
    private int lowerBound(int[] days, int targetDay) {
        int left = 0, right = days.length;
        while (left < right) {
            int mid = left + (right - left) / 2;

            // If mid is less than target, ignore the left half
            if (days[mid] < targetDay) {
                left = mid + 1;
            } else {
                // If mid is greater or equal to target, ignore the right half
                right = mid;
            }
        }
        // Return the lower bound index
        return left;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

class Solution {
public:
    vector<int> ticketDurations = {1, 7, 30}; // Durations of tickets in days
    vector<int> travelDays; // Days on which the person is traveling
    vector<int> ticketCosts; // Costs corresponding to tickets of each duration
    vector<int> dpCache; // Cache for storing results of subproblems
    int totalDays; // Total number of travel days

    // Calculates the minimum cost of all tickets needed for traveling on specific days
    int mincostTickets(vector<int>& days, vector<int>& costs) {
        totalDays = days.size(); // Set the total number of travel days
        this->travelDays = days; // Copy travel days to member variable
        this->ticketCosts = costs; // Copy ticket costs to member variable
        dpCache.assign(totalDays, -1); // Initialize the cache with -1, indicating uncomputed states
        return dfs(0); // Start the DFS traversal from day index 0
    }

    // Depth-First Search function to calculate the minimum cost of tickets starting from day index 'i'
    int dfs(int i) {
        if (i >= totalDays) return 0; // If all travel days are covered, no more cost is needed
        if (dpCache[i] != -1) return dpCache[i]; // If result is already computed, return it

        int minCost = INT_MAX; // Initialize the minimum cost to max value as we are looking for the minimum
        for (int k = 0; k < 3; ++k) { // Iterate over the three types of tickets
            // Find the next travel day index which is outside the current ticket duration
            int j = lower_bound(travelDays.begin(), travelDays.end(), travelDays[i] + ticketDurations[k]) - travelDays.begin();
            // Calculate cost and find minimum cost by trying all ticket types
            minCost = min(minCost, ticketCosts[k] + dfs(j));
        }
        dpCache[i] = minCost; // Store the result in the cache before returning
        return minCost; // Return the minimum cost for tickets starting from day index 'i'
    }
};

```

Typescript Solution

```typescript
function mincostTickets(days: number[], costs: number[]): number {
    const totalDays = days.length; // Number of days you are traveling
    const lastDay = days[totalDays - 1] + 1; // The last day of travel (plus one for indexing purposes)
    const [cost1DayPass, cost7DayPass, cost30DayPass] = costs; // Destructure costs array to name the individual pass costs
    let dp: number[] = new Array(lastDay).fill(0); // Dynamic programming array to store minimum costs up to each day

    // Iterate over each day in the dp array
    for (let day = 1; day < lastDay; day++) {
        // If the current day is among the travel days, we need to consider it for cost calculation
        let costIfBuy1DayPass = days.includes(day) ? dp[day - 1] + cost1DayPass : dp[day - 1];
        let costIfBuy7DayPass = (day > 7 ? dp[day - 7] : dp[0]) + cost7DayPass;
        let costIfBuy30DayPass = (day > 30 ? dp[day - 30] : dp[0]) + cost30DayPass;

        // The minimum cost up to the current day will be the least of the costs using any of the passes
        dp[day] = Math.min(costIfBuy1DayPass, costIfBuy7DayPass, costIfBuy30DayPass);
    }

    // Return the minimum cost on the last day of travel
    return dp[lastDay - 1];
}

```

Time and Space Complexity

The provided Python code is designed to find the minimum cost of buying tickets for a set of travel days, and the solution involves dynamic programming with memoization (`@cache`) and binary search (`bisect_left`).

Time Complexity

The time complexity of the algorithm is primarily governed by the calls to the function `dfs`. This recursive function is enhanced with memoization, which ensures that each of the `N` days is processed only once. Inside `dfs`, a binary search is performed thrice (once for each ticket type) using Python's `bisect_left`. This takes `O(log N)` time for each call. Since memoization ensures that each day is processed only once and there are 3 choices for the type of ticket at each day, the number of times `bisect_left` is called is `3N`.

Hence, the overall time complexity is `O(N * 3 * log N)`, which can be simplified to `O(N log N)`.

Space Complexity

The space complexity is determined by the storage required for memoization and the stack space used by recursive calls. The memoization (`@cache`) will store results for each of the `N` days, which gives us a space complexity of `O(N)` for memoization.

As for the recursion stack, in the worst case, the depth will be `N` (if every day is a travel day and we can buy a ticket for each single day), so the space complexity due to recursion is also `O(N)`.

Thus, the total space complexity of the algorithm is `O(N)` (where the constants are ignored, and the larger term is considered).