Minimum Cost For Tickets

Solution

You have planned some train traveling one year in advance. The days of the year in which you will travel are given as an integer array `days`. Each day is an integer from `1` to `365`.

Train tickets are sold in **three different ways**:

- a **1-day** pass is sold for `costs[0]` dollars,
- a **7-day** pass is sold for `costs[1]` dollars, and
- a **30-day** pass is sold for `costs[2]` dollars.

The passes allow that many days of consecutive travel.

- For example, if we get a **7-day** pass on day `2`, then we can travel for `7` days: `2`, `3`, `4`, `5`, `6`, `7`, and `8`.

Return *the minimum number of dollars you need to travel every day in the given list of days.*

Example 1:

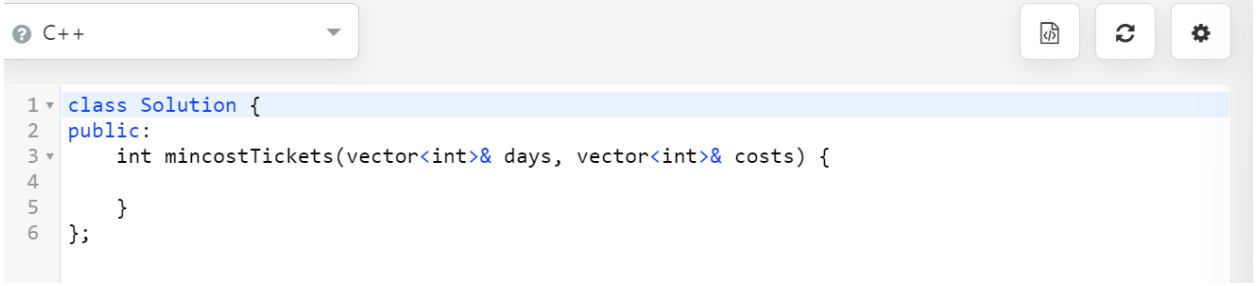
```
Input: days = [1,4,6,7,8,20], costs = [2,7,15]
Output: 11
Explanation: For example, here is one way to buy passes that lets you travel your travel plan:
On day 1, you bought a 1-day pass for costs[0] = $2, which covered day 1.
On day 3, you bought a 7-day pass for costs[1] = $7, which covered days 3, 4, ..., 9.
On day 20, you bought a 1-day pass for costs[0] = $2, which covered day 20.
In total, you spent $11 and covered all the days of your travel.
```

Example 2:

```
Input: days = [1,2,3,4,5,6,7,8,9,10,30,31], costs = [2,7,15]
Output: 17
Explanation: For example, here is one way to buy passes that lets you travel your travel plan:
On day 1, you bought a 30-day pass for costs[2] = $15 which covered days 1, 2, ..., 30.
On day 31, you bought a 1-day pass for costs[0] = $2 which covered day 31.
In total, you spent $17 and covered all the days of your travel.
```

Constraints:

- `1 <= days.length <= 365`
- `1 <= days[i] <= 365`
- `days` is in strictly increasing order.
- `costs.length == 3`
- `1 <= costs[i] <= 1000`



Solution

Approach 1: Dynamic Programming (Day Variant)

Intuition and Algorithm

For each day, if you don't have to travel today, then it's strictly better to wait to buy a pass. If you have to travel today, you have up to 3 choices: you must buy either a 1-day, 7-day, or 30-day pass.

We can express those choices as a recursion and use dynamic programming. Let's say `dp(i)` is the cost to fulfill your travel plan from day `i` to the end of the plan. Then, if you have to travel today, your cost is:

$$\mathrm{dp}(i) = \min(\mathrm{dp}(i+1) + \mathrm{costs}[0], \mathrm{dp}(i+7) + \mathrm{costs}[1], \mathrm{dp}(i+30) + \mathrm{costs}[2])$$

Java:

```
class Solution {
    int[] costs;
    Integer[] memo;
    Set<Integer> dayset;

    public int mincostTickets(int[] days, int[] costs) {
        this.costs = costs;
        memo = new Integer[366];
        dayset = new HashSet();
        for (int d: days) dayset.add(d);

        return dp(1);
```

```
    }

    public int dp(int i) {
        if (i > 365)
            return 0;
        if (memo[i] != null)
            return memo[i];

        int ans;
        if (dayset.contains(i)) {
            ans = Math.min(dp(i+1) + costs[0],
                           dp(i+7) + costs[1]);
            ans = Math.min(ans, dp(i+30) + costs[2]);
        } else {
            ans = dp(i+1);
        }

        memo[i] = ans;
        return ans;
    }
}
```

Complexity Analysis

- Time Complexity: $O(W)$, where $W = 365$ is the maximum numbered day in your travel plan.
- Space Complexity: $O(W)$.

Approach 2: Dynamic Programming (Window Variant)

Intuition and Algorithm

As in *Approach 1*, we only need to buy a travel pass on a day we intend to travel.

Now, let `dp(i)` be the cost to travel from day `days[i]` to the end of the plan. If say, `j1` is the largest index such that `days[j1] < days[i] + 1`, `j7` is the largest index such that `days[j7] < days[i] + 7`, and `j30` is the largest index such that `days[j30] < days[i] + 30`, then we have:

$$\mathrm{dp}(i) = \min(\mathrm{dp}(j1) + \mathrm{costs}[0], \mathrm{dp}(j7) + \mathrm{costs}[1], \mathrm{dp}(j30) + \mathrm{costs}[2])$$

Java:

```
class Solution {
    int[] days, costs;
    Integer[] memo;
    int[] durations = new int[]{1, 7, 30};
```

```
    public int mincostTickets(int[] days, int[] costs) {
        this.days = days;
        this.costs = costs;
        memo = new Integer[days.length];

        return dp(0);
    }

    public int dp(int i) {
        if (i >= days.length)
            return 0;
        if (memo[i] != null)
            return memo[i];

        int ans = Integer.MAX_VALUE;
        int j = i;
        for (int k = 0; k < 3; ++k) {
            while (j < days.length && days[j] < days[i] + durations[k])
                j++;
            ans = Math.min(ans, dp(j) + costs[k]);
        }

        memo[i] = ans;
        return ans;
    }
}
```

Complexity Analysis

- Time Complexity: $O(N)$, where N is the number of unique days in your travel plan.
- Space Complexity: $O(N)$.

Comments:

1. The given code is complicated, you don't need recursion to implement it.
    a. `class Solution:`
    b. `    def mincostTickets(self, days: List[int], costs: List[int]) -> int:`
    c.
    d. `        dp = [0]*(days[-1]+1)`
    e. `        days = set(days)`
    f. `        for i in range(1,len(dp)):`
    g. `            if i in days:`

```
                dp[i] =
min(dp[max(i-1,0)]+costs[0],dp[max(i-7,0)]+costs[1],dp[max(i-30,0)]
+costs[2])
            else:
                dp[i]=dp[i-1]
        return dp[-1]
```