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:

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

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

\text{dp}(i) = \min(\text{dp}(j1) + \text{costs}[0], \text{dp}(j7) + \text{costs}[1], \text{dp}(j30) + \text{costs}[2])dp(*i*)=min(dp(*j*1)+costs[0],dp(*j*7)+costs[1],dp(*j*30)+costs[2])

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