Paint House III

Solution

There is a row of `m` houses in a small city, each house must be painted with one of the `n` colors (labeled from `1` to `n`), some houses that have been painted last summer should not be painted again.

A neighborhood is a maximal group of continuous houses that are painted with the same color.

- For example: `houses = [1,2,2,3,3,2,1,1]` contains `5` neighborhoods `[{1}, {2,2}, {3,3}, {2}, {1,1}]`.

Given an array `houses`, an `m x n` matrix `cost` and an integer `target` where:

- `houses[i]`: is the color of the house `i`, and `0` if the house is not painted yet.
- `cost[i][j]`: is the cost of paint the house `i` with the color `j + 1`.

Return *the minimum cost of painting all the remaining houses in such a way that there are exactly* `target` *neighborhoods*. If it is not possible, return `-1`.

Example 1:

```
Input: houses = [0,0,0,0,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]], m = 5, n = 2, target =
Output: 9
Explanation: Paint houses of this way [1,2,2,1,1]
This array contains target = 3 neighborhoods, [{1}, {2,2}, {1,1}].
Cost of paint all houses (1 + 1 + 1 + 1 + 5) = 9.
```

Example 2:

```
Input: houses = [0,2,1,2,0], cost = [[1,10],[10,1],[10,1],[1,10],[5,1]], m = 5, n = 2, target =
Output: 11
Explanation: Some houses are already painted, Paint the houses of this way [2,2,1,2,2]
This array contains target = 3 neighborhoods, [{2,2}, {1}, {2,2}].
Cost of paint the first and last house (10 + 1) = 11.
```

Example 3:

```
Input: houses = [3,1,2,3], cost = [[1,1,1],[1,1,1],[1,1,1],[1,1,1]], m = 4, n = 3, target = 3
Output: -1
Explanation: Houses are already painted with a total of 4 neighborhoods [{3},{1},{2},{3}] diffe
```

Constraints:

- `m == houses.length == cost.length`
- `n == cost[i].length`
- `1 <= m <= 100`
- `1 <= n <= 20`
- `1 <= target <= m`
- `0 <= houses[i] <= n`
- `1 <= cost[i][j] <= 10^4`

Hide Hint #1

Use Dynamic programming.

Hide Hint #2

Define dp[i][j][k] as the minimum cost where we have k neighborhoods in the first i houses and the i-th house is painted with the color j.

C++

```cpp
class Solution {
public:
    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int target) {

    }
};
```

Solution

Overview

There are `m` houses, of which some have been already painted, and others still need to be painted with one of `n` colors. We need to paint these houses to minimize the cost while ensuring that there are exactly `target` number of neighborhoods. A continuous group of houses with the same color is considered a single neighborhood.

There are two characteristics of this problem that we should take note of at this time. First, as we iterate over houses, we must decide what color to paint each unpainted house. The optimal choice will depend on how we have painted the previous houses, as this choice can affect the total number of neighborhoods. In other words, each decision we make is affected by the previous decisions we have made. Second, the problem is asking to minimize the cost of painting all unpainted houses. These two characteristics suggest that we may be able to solve this problem using dynamic programming.

Approach 1: Top-Down Dynamic Programming

Intuition

Let's start with the first house. If it is already painted, then we have nothing to do and can move on to the next house. Otherwise, we can choose any color, from `1` to `n`, and spend the corresponding cost to paint the house that color. Note that we need to count the number of neighborhoods as well. So after visiting a house, we need to update the number of neighborhoods. Let's say the current number of neighborhoods is equal to `neighborhoodCount`. When we reach a particular house, we will compare its color with the previous house color. If the colors don't match, we will increment the value `neighborhoodCount` by `1` and recursively move on to the next house with updated values.

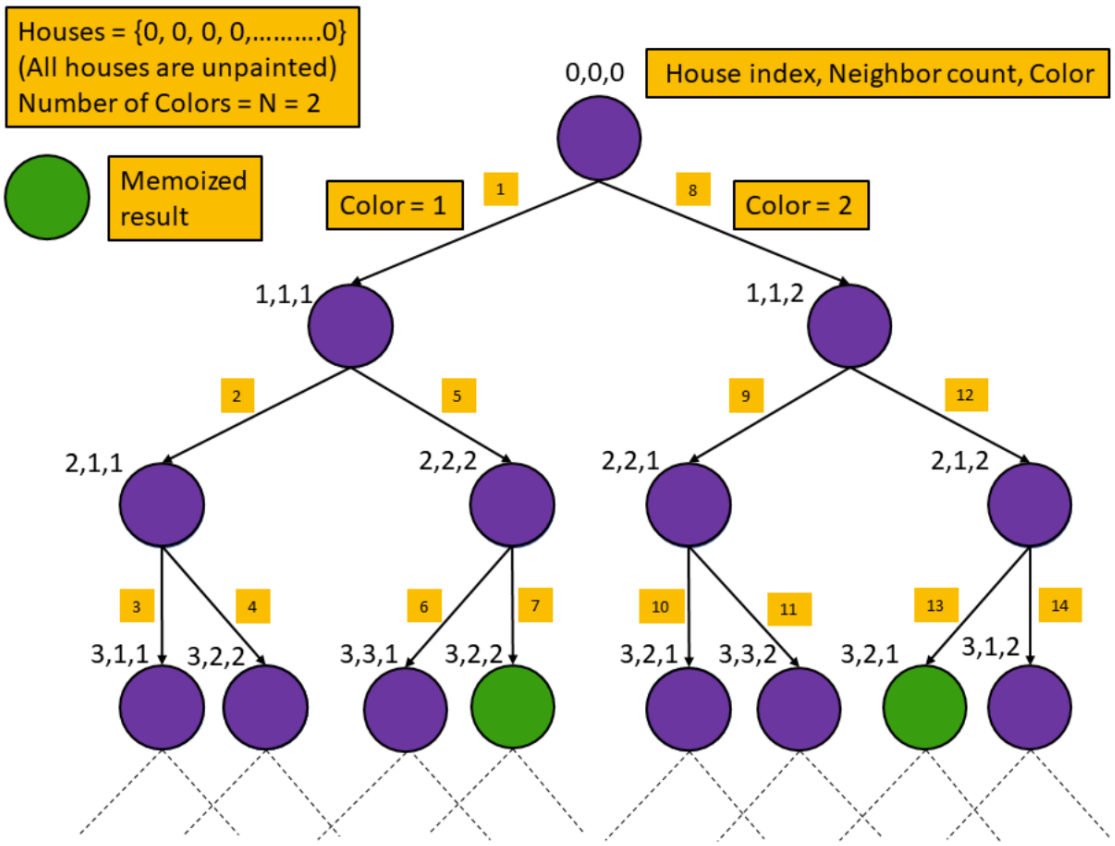
This way, we can try every possible color for unpainted houses and find the cost. After traversing all of the houses, if the number of neighborhoods equals `target`, then we can compare the cost with the minimum cost we have achieved so far and update it accordingly.

For this recursive approach, what are the parameters that we need to track? The first parameter is the index of the house that we are currently considering as we traverse the houses. Since the number of neighborhoods must equal `target`, we also need a variable to store the count of the neighborhoods. Finally, we must also keep track of the previous house color, and when the current house color does not match the previous house color, we know that we are at the start of a new neighborhood.

Hence, we need to keep track of three things:

1. The index of the house that we are currently considering.
2. The color of the previous house.
3. The current number of neighborhoods.

This recursive approach will have repeated subproblems; this can be observed in the figure below. Notice the green nodes are repeated subproblems signifying that we have already solved these subproblems before.



To avoid recalculating results for previously seen subproblems, we will memoize the result for each subproblem. So the next time we need to calculate the result for the same set of parameters `{current index, neighborhood count, previous house color}`, we can simply look up the result in constant time instead of recalculating the result.

Algorithm

1. Start with:
    - The current index `currIndex` as `0`; this is the index of the house we are currently considering.
    - The current number of neighborhoods `neighborhoodCount` as `0`.
    - The color of the previous house `prevHouseColor` as `0` (we choose to initialize this value as `0` to guarantee that the first house will always be counted as the start of a new neighborhood).
2. If the house at `currIndex` is already painted (not equal to `0`), we can recursively move on to the next house with the updated value of `neighborhoodCount`. Remember to increment `neighborhoodCount` if the color of the house at `currIndex` does not equal `prevHouseColor`. Also, update `prevHouseColor` to the color of the current house.
3. If the house is not painted, iterate over colors from `1` to `n`, and for each color, try painting the house at `currIndex` with that color, then recursively move on to the next house with updated values. Store the minimum cost corresponding to any color from `1` to `n` in the variable `minCost`.
4. Return the value `minCost` and also store it in the memoization table `memo` corresponding to the current state, which is defined by `currIndex`, `neighborhoodCount`, and `prevHouseColor`.
5. Base cases:
    - If we have traversed over all the houses i.e., `currIndex == m`, we will return the cost `0` if the `neighborhoodCount` is equal to `target`, otherwise we return the value `MAX_COST` which is the maximum possible cost plus `1`.
    - If the number of neighborhoods `neighborhoodCount` is more than the target neighborhoods `neighborhoodCount > targetCount`, then the answer is not possible. Hence we return `MAX_COST`.

Implementation

C++:

```
class Solution {
public:
    // Assign the size as per maximum value for different params
    int memo[100][100][21];
    // Maximum cost possible plus 1
    const int MAX_COST = 1000001;

    int findMinCost(vector<int>& houses, vector<vector<int>>& cost, int targetCount, int
currIndex,
                    int neighborhoodCount, int prevHouseColor) {
        if (currIndex == houses.size()) {
            // If all houses are traversed, check if the neighbor count is as expected or not
            return neighborhoodCount == targetCount ? 0 : MAX_COST;
        }

        if (neighborhoodCount > targetCount) {
            // If the neighborhoods are more than the threshold, we can't have target neighborhoods
            return MAX_COST;
        }
```

```
        // We have already calculated the answer so no need to go into recursion
        if (memo[currIndex][neighborhoodCount][prevHouseColor] != -1) {
            return memo[currIndex][neighborhoodCount][prevHouseColor];
        }

        int minCost = MAX_COST;
        // If the house is already painted, update the values accordingly
        if (houses[currIndex]) {
            int newNeighborhoodCount = neighborhoodCount + (houses[currIndex] !=
prevHouseColor);
            minCost =
                findMinCost(houses, cost, targetCount, currIndex + 1, newNeighborhoodCount,
houses[currIndex]);
        } else {
            int totalColors = cost[0].size();

            // If the house is not painted, try every possible color and store the minimum cost
            for (int color = 1; color <= totalColors; color++) {
                int newNeighborhoodCount = neighborhoodCount + (color != prevHouseColor);
                int currCost = cost[currIndex][color - 1]
                    + findMinCost(houses, cost, targetCount, currIndex + 1, newNeighborhoodCount,
color);
                minCost = min(minCost, currCost);
            }
        }

        // Return the minimum cost and also storing it for future reference (memoization)
        return memo[currIndex][neighborhoodCount][prevHouseColor] = minCost;
    }

    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int target) {
        // Marking all values to -1 to denote that we don't have the answer ready for these params
yet
        memset(memo, -1, sizeof(memo));
        int answer = findMinCost(houses, cost, target, 0, 0, 0);

        // Return -1 if the answer is MAX_COST as it implies no answer possible
        return answer == MAX_COST ? -1 : answer;
    }
};
```

Complexity Analysis

Here, M is the number of houses, N is the number of colors and T is the number of target neighborhoods.

- Time complexity: $O(M \cdot T \cdot N^2)$

  Each state is defined by the values `currIndex`, `neighborhoodCount`, and `prevHouseColor`. Hence, there will be $M \cdot T \cdot N$ possible states, and in the worst-case scenario, we must visit most of the states to solve the original problem. Each recursive call requires $O(N)$ time as we might need to iterate over all the colors. Thus, the total time complexity is equal to $O(M \cdot T \cdot N^2)$.

- Space complexity: $O(M \cdot T \cdot N)$

  The memoization results are stored in the table `memo` with size $M \cdot T \cdot N$. Also, stack space in the recursion is equal to the maximum number of active functions. The maximum number of active functions will be at most M i.e., one function call for every house. Hence, the space complexity is $O(M \cdot T \cdot N)$.

Approach 2: Bottom-Up Dynamic Programming

Intuition

In the previous approach, the recursive calls incurred stack space. We can avoid this by applying the same approach iteratively, which is often faster than the top-down approach. We will follow a similar approach to the previous one. However, this time we will iterate over the states by starting from the base case and ending at the initial query.

Suppose we want to find the minimum cost to paint the first `house` number of houses with `neighborhoods` number of neighborhoods. Let's see how we can break this problem into subproblems. The house at the index `house` can be painted with one of the `n` colors, so we iterate over the colors, and for each color `color`, there will be one of the two following scenarios:

- The house at index `house` is already painted, and the color of the house does not match `color`.
    - In this case, we do not need to do anything since we cannot paint a house that is already painted.
- The house at index `house` is not painted yet, or it is already painted the color `color`.
    - In this case, if the house is not painted, we will paint the house at index `house` with the color `color` for the corresponding cost. If the house is already painted, the corresponding cost will be `0`. Since we are trying to find the minimum cost for the state defined by { `house`, `neighborhoods`, and `color` }, we must iterate over all of the color options (`prevColor`) for the previous house to do so.
        - If the `color` and `prevColor` are not the same, the subproblem will be { `house - 1`, `neighborhoods - 1`, `prevColor` } because whenever two adjacent houses are not the same color, this marks the start of a new neighborhood.
        - If the `color` and `prevColor` are the same, the subproblem will be { `house - 1`, `neighborhoods`, `color` }. This is because the neighborhoods won't change as the house at index `house` will be added to the previous neighborhood.

The only case where we don't need to break the problem into subproblems (base cases) is when we only have the first house. For this base case, the value of the neighborhood will be `1`, and we can assign the cost for each color from `1` to `n` according to the cost to paint the house at index `0` that color.

Once we have found the cost for all possible combinations of houses, color, and neighborhoods, we can find the minimum cost to paint all the houses with the `target` number of neighborhoods. This value will be present in the array `memo[m - 1][target]`, and the minimum value in this array would be our answer. That particular index represents the color that we should use to paint the last house (if it was not painted originally).

Algorithm

1. Initialize the values for base case i.e., `house = 0`, `neighborhoods = 1`. Iterate over the colors from `1` to `n` and assign the corresponding cost if the house is not painted. Otherwise, if the house is already painted with the same color, assign the cost as `0`.
2. Iterate over house index `house` from `1` to `m - 1`, and neighborhoods count `neighborhoods` from `1` to the minimum of `(house + 1, target)` (for `0` indexed `house`, `house + 1` is the maximum number of neighborhoods possible)), and color for the current house, from `1` to `n`. For each of these iterations:
    - If the house at index `house` is already painted and the color is not the same as `color`, then continue, since we never repaint a painted house.
    - Initialize the cost for current parameters `currCost` to `MAX_COST` which is the maximum possible cost plus `1`.
    - Iterate over the color options for the previous house `prevColor` from `1` to `n`, for each `prevColor`:
        - If the `color` and `prevColor` are different, assign the minimum of `currCost` and `memo[house - 1][neighborhoods - 1][prevColor - 1]` to `currCost`.
        - If the `color` and `prevColor` match, assign the minimum of `currCost` and `memo[house - 1][neighborhoods][color - 1]` to `currCost`.
        - **Note:** We used `color - 1` and `prevColor - 1` instead of `color` and `prevColor` respectively as we are using `0` based indexing.
    - Assign the cost to paint the current house at index `house` with `color` in the variable `costToPaint`.
    - Assign the value `currCost + costToPaint` to `memo[house][neighborhoods][color - 1]`.
3. Find the minimum cost among all the options to paint the last house with `target` number of neighborhoods, store it in the `minCost`.
4. Return `minCost`.

Implementation

C++:

```
class Solution {
public:
    // Maximum cost possible plus 1
    const int MAX_COST = 1000001;

    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int target) {
        // Initialize memo array
        vector<vector<vector<int>>> memo(m, vector<vector<int>>(target + 1, vector<int>(n,
MAX_COST)));

        // Initialize for house 0, neighborhoods will be 1
        for (int color = 1; color <= n; color++) {
            if (houses[0] == color) {
                // If the house has same color, no cost
                memo[0][1][color - 1] = 0;
```

```
        } else if (!houses[0]) {
            // If the house is not painted, assign the corresponding cost
            memo[0][1][color - 1] = cost[0][color - 1];
        }
    }

    for (int house = 1; house < m; house++) {
        for (int neighborhoods = 1; neighborhoods <= min(target, house + 1); neighborhoods++)
{
            for (int color = 1; color <= n; color++) {
                // If the house is already painted, and color is different
                if (houses[house] && color != houses[house]) {
                    // Cannot be painted with different color
                    continue;
                }

                int currCost = MAX_COST;
                // Iterate over all the possible color for previous house
                for (int prevColor = 1; prevColor <= n; prevColor++) {
                    if (prevColor != color) {
                        // Decrement the neighborhood as adjacent houses has different color
                        currCost = min(currCost, memo[house - 1][neighborhoods - 1][prevColor - 1]);
                    } else {
                        // Previous house has the same color, no change in neighborhood count
                        currCost = min(currCost, memo[house - 1][neighborhoods][color - 1]);
                    }
                }

                // If the house is already painted cost to paint is 0
                int costToPaint = houses[house] ? 0 : cost[house][color - 1];
                memo[house][neighborhoods][color - 1] = currCost + costToPaint;
            }
        }
    }

    int minCost = MAX_COST;
    // Find the minimum cost with m houses and target neighborhoods
    // By comparing cost for different color for the last house
    for (int color = 1; color <= n; color++) {
        minCost = min(minCost, memo[m - 1][target][color - 1]);
    }

    // Return -1 if the answer is MAX_COST as it implies no answer possible
    return minCost == MAX_COST ? -1 : minCost;
```

```
    }
};
```

Complexity Analysis

Here, M is the number of houses, N is the number of colors and T is the number of target neighborhoods.

- Time complexity: $O(M \cdot T \cdot N^2)$

  Each state is defined by the values `house`, `neighborhoods`, and `color`. Hence, there will be $M \cdot T \cdot N$ possible states, and in the worst-case scenario, we must visit most of the states to solve the original problem. Each state (subproblem) requires $O(N)$ time as we iterate over all the colors for `prevColor`. Thus, the total time complexity is equal to $O(M \cdot T \cdot N^2)$.

- Space complexity: $O(M \cdot T \cdot N)$

  The results are stored in the table `memo` with size $M \cdot T \cdot N$. Hence, the space complexity is equal to $O(M \cdot T \cdot N)$.

Approach 3: Bottom-Up Dynamic Programming (Space Optimized)

Intuition

In the previous approach, to find the minimum cost to paint the current house the color `color`, we use the cost that has been already calculated for the previous house (the hosue at index `house - 1`). Even though the recurrence relation only uses the cost corresponding to the previous house, we allocate enough space to store the cost for every house, which is unnecessary. Observing the code for the previous approach, we can see that we either used `memo[house - 1][neighborhoods - 1][prevColor - 1]` or `memo[house - 1][neighborhoods][color - 1]` to find the cost for the house at index `house`. This means we only referred to the cost for the previous house.

Therefore, in this approach, we will only store the costs that correspond to the previous house. With the help of the results for the previous house, we will find the cost for the current house and then store it to be used as the previous house result by the next house. This way, we don't need to store the result for every house simultaneously; instead, we can just store the result for the current and previous houses.

Algorithm

1. Store the results for the base cases in `prevMemo`. The results are the cost for each color from `1` to `n` when we are at the first house (`house = 0`) and the number of neighborhoods is `1`.
2. Iterate over house index `house` from `1` to `m - 1`,
    - Create a new table `memo`; this table will store the results for the current house.
    - Iterate over neighborhoods count `neighborhoods` from `1` to minimum of `(house + 1, target)` and iterate over color for the current house `color` from `1` to `n`. For each iteration:
        - If the house at index `house` is already painted and the color is not the same as `color`, then continue, since we never repaint a painted house.
        - Initialize the cost for current options `currCost` to `MAX_COST` which is the maximum possible cost plus `1`.
        - Iterate over the color option for the previous house `prevColor` from `1` to `n`, for each `prevColor`:
            - If the `color` and `prevColor` are different, assign the minimum of `currCost` and `prevMemo[neighborhoods - 1][prevColor - 1]` to `currCost`.
            - If the `color` and `prevColor` match, assign the minimum of `currCost` and `prevMemo[neighborhoods][color - 1]` to `currCost`.
    - Assign the cost to paint the current house at index `house` with `color` in the variable `costToPaint`.
    - Assign the value `currCost + costToPaint` to `memo[neighborhoods][color - 1]`.
    - Update the table `prevMemo` to store the results for the current house by assigning it to `memo`.
3. Find the minimum cost among all the options to paint the last house with `target` number of neighborhoods, store it in the `minCost`.
4. Return `minCost`.

Implementation

C++:

```
class Solution {
public:
    // Maximum cost possible plus 1
    const int MAX_COST = 1000001;

    int minCost(vector<int>& houses, vector<vector<int>>& cost, int m, int n, int target) {
        // Initialize prevMemo array
        vector<vector<int>> prevMemo(target + 1, vector<int>(n, MAX_COST));

        // Initialize for house 0, neighborhood will be 1
        for (int color = 1; color <= n; color++) {
            if (houses[0] == color) {
                // If the house has same color, no cost
                prevMemo[1][color - 1] = 0;
```

```
        } else if (!houses[0]) {
            // If the house is not painted, assign the corresponding cost
            prevMemo[1][color - 1] = cost[0][color - 1];
        }
    }

    for (int house = 1; house < m; house++) {
        // Initialize memo array
        vector<vector<int>> memo(target + 1, vector<int>(n, MAX_COST));

        for (int neighborhoods = 1; neighborhoods <= min(target, house + 1); neighborhoods++)
{
            for (int color = 1; color <= n; color++) {
                // If the house is already painted, and color is different
                if (houses[house] && color != houses[house]) {
                    // Cannot be painted with different color
                    continue;
                }

                int currCost = MAX_COST;
                // Iterate over all the possible color for previous house
                for (int prevColor = 1; prevColor <= n; prevColor++) {
                    if (prevColor != color) {
                        // Decrement the neighborhood as adjacent houses has different color
                        currCost = min(currCost, prevMemo[neighborhoods - 1][prevColor - 1]);
                    } else {
                        // Previous house has the same color, no change in neighborhood count
                        currCost = min(currCost, prevMemo[neighborhoods][color - 1]);
                    }
                }

                // If the house is already painted cost to paint is 0
                int costToPaint = houses[house] ? 0 : cost[house][color - 1];
                memo[neighborhoods][color - 1] = currCost + costToPaint;
            }
        }
        // Update the table to have the current house results
        prevMemo = memo;
    }

    int minCost = MAX_COST;
    // Find the minimum cost with m houses and target neighborhoods
    // By comparing cost for different color for the last house
    for (int color = 1; color <= n; color++) {
```

```
            minCost = min(minCost, prevMemo[target][color - 1]);
        }

        // Return -1 if the answer is MAX_COST as it implies no answer possible
        return minCost == MAX_COST ? -1 : minCost;
    }
};
```

Complexity Analysis

Here, M is the number of houses, N is the number of colors and T is the number of target neighborhoods.

- Time complexity: $O(M \cdot T \cdot N^2)$

  We are iterating over the houses from 1 to M and for each house, store the results in the table `memo` by iterating over each neighbor and color. Therefore, we have $T \cdot N$ states for each house, and each such state will take $O(N)$ operations to iterate over the `prevColor` options. Hence the total time complexity is $O(M \cdot T \cdot N^2)$.

- Space complexity: $O(T \cdot N)$

  The results are stored in the arrays `memo` and `prevMemo`, each with a size of $T \cdot N$. Hence, the space complexity equals $O(T \cdot N)$.