Paint House II

Solution

There are a row of `n` houses, each house can be painted with one of the `k` colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by an `n x k` cost matrix costs.

- For example, `costs[0][0]` is the cost of painting house `0` with color `0`; `costs[1][2]` is the cost of painting house `1` with color `2`, and so on...

Return *the minimum cost to paint all houses*.

Example 1:

```
Input: costs = [[1,5,3],[2,9,4]]
Output: 5
Explanation:
Paint house 0 into color 0, paint house 1 into color 2. Minimum cost: 1 + 4 = 5;
Or paint house 0 into color 2, paint house 1 into color 0. Minimum cost: 3 + 2 = 5.
```

Example 2:

```
Input: costs = [[1,3],[2,4]]
Output: 5
```

Constraints:

- `costs.length == n`
- `costs[i].length == k`
- `1 <= n <= 100`
- `2 <= k <= 20`
- `1 <= costs[i][j] <= 20`

Follow up: Could you solve it in `O(nk)` runtime?

Solution

Paint House II is a follow up question of Paint House. In the original Paint House problem, `k` was always `3`. In this problem, `k` is no longer fixed and instead can be any non-negative integer.

If you haven't yet attempted the original Paint House question and are having trouble with this question, go attempt Paint House first and come back. There is also an in-depth Paint House Solution Article. This solution article will assume you are already comfortable with the memoization and dynamic programming solutions for Paint House.

Approach 1: Memoization

Intuition

Remembering that we already know how to solve this problem using memoization when `k = 3` (check the Paint House Solution Article) if you can't remember how), let's think through some of the other possible values of `k`.

For this explanation, we'll call a way of painting the houses *valid* if, and only if, there are no adjacent houses painted the same color. We'll call an input *valid* if it is possible to paint the houses in a *valid* way. **The test cases here on Leetcode are all valid inputs**. In an interview however, you'd need to ensure that it is safe to assume that the input is always valid though.

If `k = 0`, then this means we have no colors. If there are no colors, it's probably reasonable to assume there are no houses either, i.e. `n = 0`. In other words, the input is `[]`. **For this question here on Leetcode, this is a safe assumption**. In an interview though it could be a good idea to ask the interviewer whether or not the input is *guaranteed* to be valid. For example, could you get a test case such as `[[],[],[],[]]`? This would be `k = 0` and `n = 4`. Of course, this case doesn't make much sense, because we are supposed to be painting houses, but can't with no paint. Either you'd be told it could never happen, or that you needed to do something special for it, such as returning `-1`.

If `k = 1` (all houses have to be the same color), then it's probably safe to assume that `n = 1`. Otherwise, the problem would be impossible to solve without breaking the adjacent color rule. Again, this is a safe assumption here, but do consider asking the interviewer whether or not you could get an invalid input that had `k = 1` and `n > 1`. So, assuming that `k = 1` and `n = 1`, the total cost will be the cost of painting that one house the only color available.

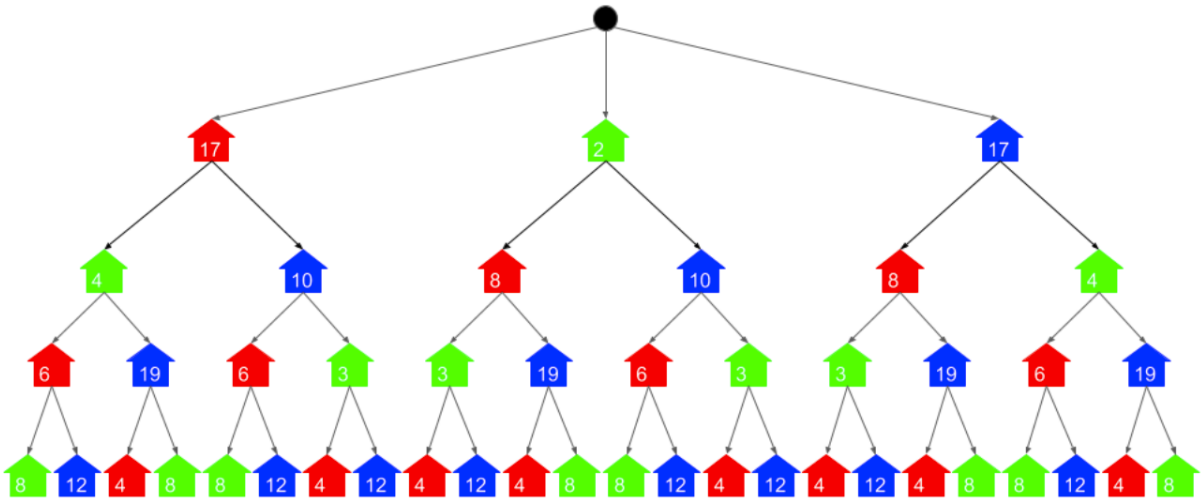
If `k = 2` (there are two colors), then we know the problem is always solvable, because we can simply paint the houses alternating colors. For example, when `n = 5` and `k = 2`, here are the only 2 valid ways of painting the houses. Anything else would be invalid.

The answer will be the one that leads to the lowest cost. It'd be easy to check both.

When `k = 3`, the problem is equivalent to Paint House. In the Solution Article for that question, we worked through an example where `n = 4`.

```
[[17, 2, 17], [8, 4, 10], [6, 3, 19], [4, 8, 12]]
```

A good way to visualize all of the valid painting permutations is to use a tree. Each root-to-leaf path represents one valid way of painting the houses.



The cheapest cost of painting the houses is, therefore, the root-to-leaf path with the lowest total sum of its nodes. This animation shows the algorithm we used to solve this problem for Paint House.

Video:

Luckily, we didn't actually need to create the tree itself—there is a simpler way using recursion.

Say we have a `paint` function that takes 2 parameters: a house number and a color to paint that house. The output is the ***total* cost of painting that house and all the ones after it**. For example `paint(1, red)` would be the cost of painting house `1` red, along with the cost of painting the houses after it (taking into account restrictions caused by painting house `1` red).

Therefore the cheapest way of painting all the houses can be expressed as follows, where `0` is the first house.

`min(paint(0, "red"), paint(0, "green"), paint(0, "blue"))`

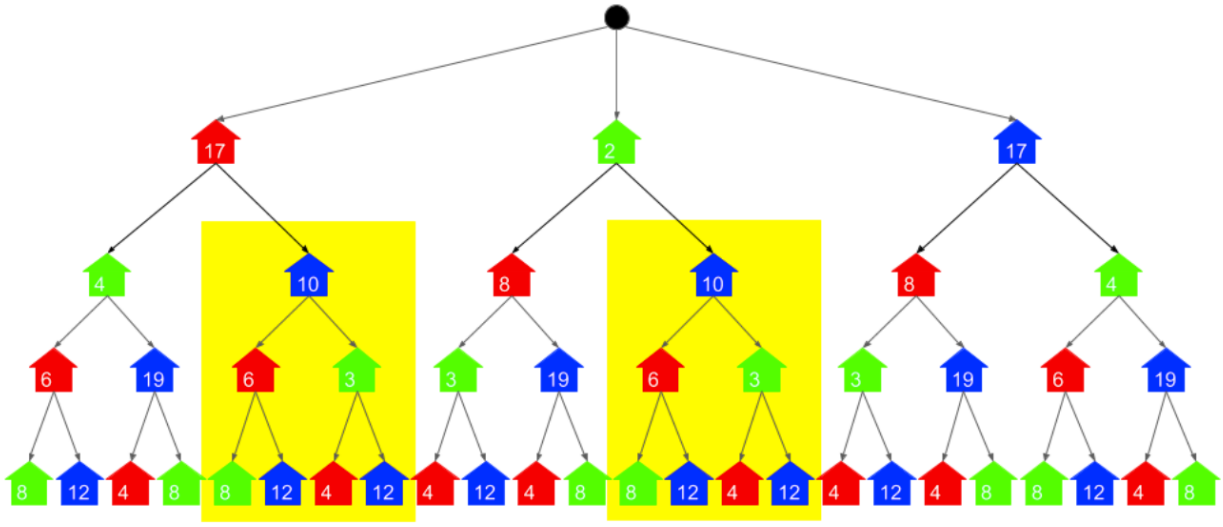
The `paint` function has a recursive implementation. `costs` refers to the *input* table.

```
def paint(i, color):
    ### BASE CASE ###
    if i is the last house number:
        return costs[i][color]
    ### RECURSIVE CASE ###
    lowest_cost = Infinity
    for each next_color in ["red", "green", "blue"]:
       if next_color != color: # No adjacent houses can be same color.
           this_cost = costs[i][color] + paint(i + 1, next_color) # <- Recursive call
           lowest_cost = min(lowest_cost, this_cost)
    return lowest_cost
```

The **base case** is where `i` refers to the *last house*. Painting the last house a particular color can be obtained from the `costs` table.

The **recursive case** is where we also need to consider the houses after `i`. It is obtained by looking up the cost of painting house `i` the given color (in the `costs` table) and then by determining the cost of painting the houses after it. The cost of painting the houses after requires making recursive paint calls to determine the cost of painting house `i + 1` each of the 2 other colors and then finding the minimum of those 2 values.

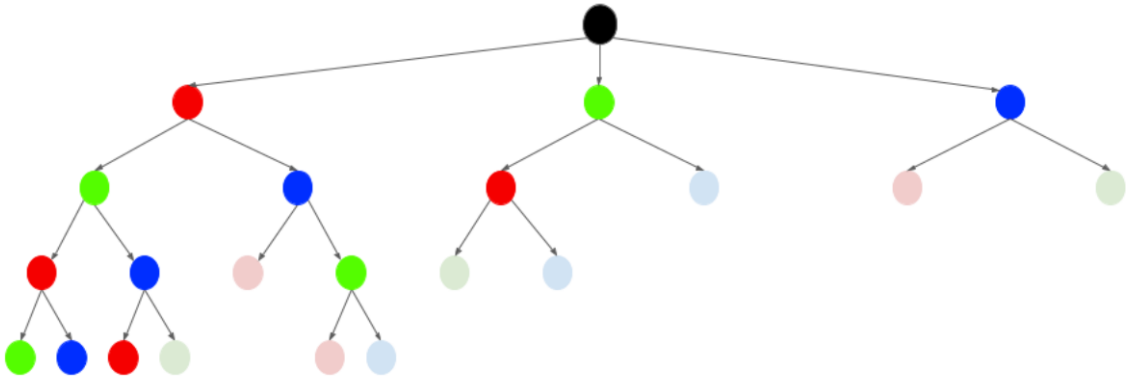
This algorithm is inefficient though. There is a lot of repetition in the tree, meaning we're doing the same calculations over and over again. For example, the total cost of painting the *second* house blue will be the same regardless of whether the first house was red or green. For example, both these branches of the tree are identical.



This should also be apparent from the definition of the `paint` function. The parameters are simply a house number and a color. It doesn't require any information about *where* exactly in the tree it is.

To solve this problem, we add **memoization** to the `paint` function. Recall that memoization is where before returning an answer, the recursive function writes the answer into a dictionary with the input parameters as the key and the answer as the value. Then before doing a calculation, it checks whether or not that particular calculation has already been done. For the example above, the function would only calculate the cost of painting the second house blue *once*, and then the second time it would look it up in the dictionary.

Here is a visualization that shows the calculations that need to be done when using memoization. The brighter circles show where the function body runs, and the duller circles show where a lookup was done and the function immediately returned. These are the only times the function is called.



If this explanation wasn't thorough enough for you, then please check out the full Paint House Solution Article. In that article I go into a lot more depth about memoization and how this algorithm was derived.

The `k > 3` case is really no different to this. The only difference is that instead of only considering 2 possible colors for the *next* house, we're considering `k - 1` colors (all colors except for the color of the current house).

Algorithm

Unlike the pseudocode above, we don't need to worry about the names of the colors. Instead, they are represented by numbers between `0` and `k - 1`. Additionally, we're also using memoization (`lru_cache` in Python, and a dictionary in Java).

Implementation

```
class Solution {

    private int n;
    private int k;
    private int[][] costs;
    private Map<String, Integer> memo;

    public int minCostII(int[][] costs) {
        if (costs.length == 0) return 0;
        this.k = costs[0].length;
        this.n = costs.length;
        this.costs = costs;
        this.memo = new HashMap<>();
        int minCost = Integer.MAX_VALUE;
```

```
    for (int color = 0; color < k; color++) {
        minCost = Math.min(minCost, memoSolve(0, color));
    }
    return minCost;
}

private int memoSolve(int houseNumber, int color) {

    // Base case: There are no more houses after this one.
    if (houseNumber == n - 1) {
        return costs[houseNumber][color];
    }

    // Memoization lookup case: Have we already solved this subproblem?
    if (memo.containsKey(getKey(houseNumber, color))) {
        return memo.get(getKey(houseNumber, color));
    }

    // Recursive case: Determine the minimum cost for the remainder.
    int minRemainingCost = Integer.MAX_VALUE;
    for (int nextColor = 0; nextColor < k; nextColor++) {
        if (color == nextColor) continue;
        int currentRemainingCost = memoSolve(houseNumber + 1, nextColor);
        minRemainingCost = Math.min(currentRemainingCost, minRemainingCost);
    }
    int totalCost = costs[houseNumber][color] + minRemainingCost;
    memo.put(getKey(houseNumber, color), totalCost);
    return totalCost;
}

// Convert a house number and color into a simple string key for the memo.
```

```
    private String getKey(int n, int color) {
        return String.valueOf(n) + " " + String.valueOf(color);
    }
}
```

Complexity Analysis

- Time complexity : $O(n \cdot k^2)$.

  Determining the total time complexity of a recursive memoization algorithm requires looking at how many calls are made to the `paint` function, and how much each call costs (remember that the memoization lookups are $O(1)$). The function is called once for each possible pair of house number and color. This gives $n \cdot k$ calls. Then, each call has a loop that loops over each of the k colors. Therefore, we have $n \cdot k \cdot k = n \cdot k^2$ which is $O(n \cdot k^2)$.

  The part outside of the recursive function is $O(k)$ and therefore does not impact the overall complexity.

- Space complexity : $O(n \cdot k)$.

  There are 2 different places memory is being used that we need to consider.

  Firstly, the memoization is storing the answers for each pair of house number and color. There are $n \cdot k$ of these, and so $O(n \cdot k)$ memory used.

  Secondly, we need to consider the memory used on the run-time stack. In the worst case, there's a stack frame for each house number on the stack. This is a total of $O(n)$.

  The $O(n)$ is insignficant to the $O(n \cdot k)$, so we're left with a total of $O(n \cdot k)$.

Approach 2: Dynamic Programming

Intuition

Let's look at a bigger example now, and view the problem in a different way to how we did before. For this example, `k = 6` and `n = 5`.

`[[10, 6, 16, 25, 7, 28], [7, 16, 18, 30, 16, 25], [8, 26, 6, 22, 26, 19], [10, 23, 14, 17, 23, 9], [12, 14, 27, 7, 8, 9]]`

And here is a diagram of the input grid.

	0 (Red)	**1** (Green)	**2** (Blue)	**3** (Yellow)	**4** (Purple)	**5** (Orange)
0	10	6	16	25	7	28
1	7	16	18	30	16	25
2	8	26	6	22	26	19
3	10	23	14	17	23	9
4	12	14	27	7	8	9

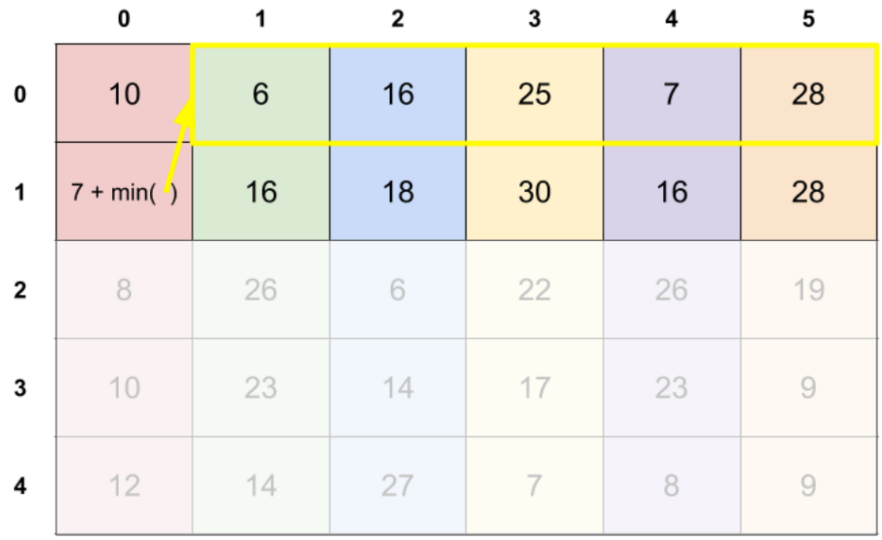
Each row represents the different colors a house could be. *Remember that the colors are represented by numbers.* The actual colors are only to make the table easier to read.

The problem we're trying to solve is equivalent to the following: **pick exactly one number from each row** such that the **sum of those numbers is minimized**. Because 2 adjacent houses cannot be the same color, **adjacent rows must be picked from different columns**. This is a straightforward variant of one of those "classic" minimum-path-in-a-grid dynamic programming problems.

The way that we solve it is to iterate over the cells and determine what the cheapest way of getting to that cell is. We'll work from top to bottom.

To begin with, we say the first row (house 0) is already completed. We don't need to make any changes to it.

Then, for each cell in the second row, we work out the cheapest way of getting to it from the first row is. For example, to get to `[1][red]` we have to go through any of the non-red cells from the row above. We want to go through the minimum.



We show our decision by updating `[1][red]` to `7 + 6 = 13`.

We can repeat this for the rest of the second row, and then work down each of the remaining rows.

Here's an animation of the algorithm being carried out.

When we're finished, the final answer is the **minimum value in the last row**.

Algorithm

We'll do this in the same way we did in the animation above—an in-place algorithm that over-writes the input grid.

Implementation

```
class Solution {
    public int minCostII(int[][] costs) {

        if (costs.length == 0) return 0;
        int k = costs[0].length;
        int n = costs.length;

        for (int house = 1; house < n; house++) {
            for (int color = 0; color < k; color++) {
                int min = Integer.MAX_VALUE;
                for (int previousColor = 0; previousColor < k; previousColor++) {
                    if (color == previousColor) continue;
                    min = Math.min(min, costs[house - 1][previousColor]);
                }
                costs[house][color] += min;
            }
        }

        // Find the minimum in the last row.
        int min = Integer.MAX_VALUE;
        for (int c : costs[n - 1]) {
            min = Math.min(min, c);
        }
        return min;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \cdot k^2)$.

  We iterate over each of the $n \cdot k$ cells. For each of the cells, we're finding the minimum of the k values in the row above, excluding the one that is in the same column. This operation is $O(k)$. Multiplying this out, we get $O(n \cdot k^2)$.

- Space complexity : $O(1)$ if done in-place, $O(n \cdot k)$ if input is copied.

  We're not creating any new data structures in the code above, and so it has a space complexity of $O(1)$. This is, however, overwriting the given input, which might not be ideal in some situations.

  If we don't want to overwrite the input, we could instead create a copy of it first and then do the calculations in the copy. This will require an additional $O(n \cdot k)$ space.

Approach 3: Dynamic Programming with O(k) additional Space.

Intuition

Implementing the algorithm in-place meant that we only needed $O(1)$ additional space. This, however required modifying the input, which could be a problem in some situations.

The easiest solution is to make a copy of the input array and then do the calculations in that instead. This would require $O(n \cdot k)$ additional space.

There is a way that uses less space though. We're only ever working with 2 rows at a time: the current row, and the row before it. The rows before that are never looked at again, and the rows after are still the same as the input array. Therefore, we can take advantage of this to only use $O(k)$ space.

Algorithm

Instead of writing the updated costs into the input array, the algorithm writes them into a k-length array. The k-length array from the previous row is held onto in-order to do these calculations.

Implementation

```
class Solution {


  public int minCostII(int[][] costs) {


    if (costs.length == 0) return 0;
    int k = costs[0].length;
    int n = costs.length;


    int[] previousRow = costs[0];
```

```
        for (int house = 1; house < n; house++) {
            int[] currentRow = new int[k];
            for (int color = 0; color < k; color++) {
                int min = Integer.MAX_VALUE;
                for (int previousColor = 0; previousColor < k; previousColor++) {
                    if (color == previousColor) continue;
                    min = Math.min(min, previousRow[previousColor]);
                }
                currentRow[color] += costs[house][color] += min;
            }
            previousRow = currentRow;
        }

        // Find the minimum in the last row.
        int min = Integer.MAX_VALUE;
        for (int c : previousRow) {
            min = Math.min(min, c);
        }
        return min;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \cdot k^2)$.

  Same as above.

- Space complexity : $O(k)$.

  The previous row and the current row are represented as k-length arrays.

  This approach does *not* modify the input grid.

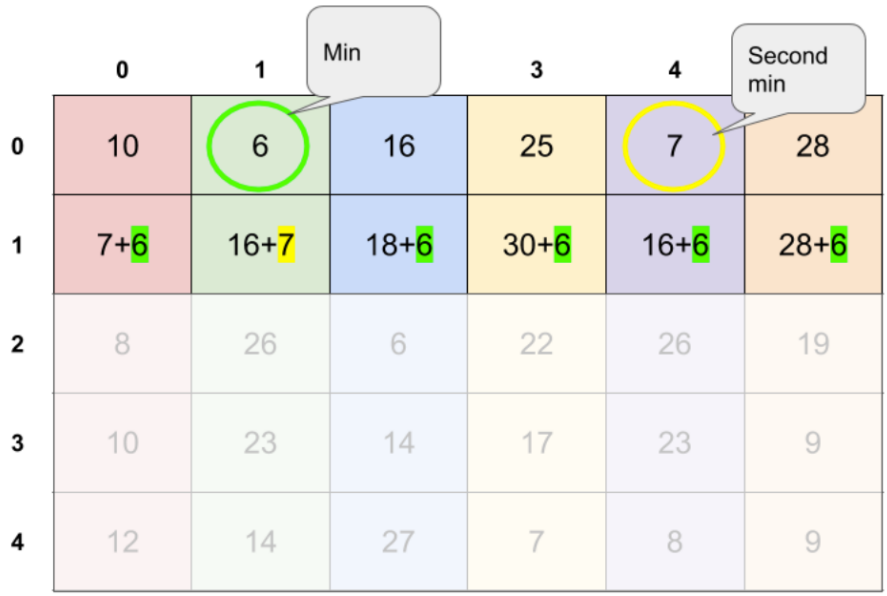
Approach 4: Dynamic programming with Optimized Time

Intuition

Despite Paint House II being listed as a hard question, and the problem statement listing $O(n \cdot k)$ time as a "follow up", you'd possibly be expected to come up with this solution at top companies as it's still a fairly basic dynamic programming algorithm. You should, therefore, ensure you're comfortable with this approach and could identify and apply similar observations in other dynamic programming problems. At the very least, it'll make you look awesome!

So far, all of our approaches have had a $O(n \cdot k^2)$ time complexity. This is because calculating the new value for each of the $O(n \cdot k)$ cells required looking at each of the k cells in the row immediately below.

However, we don't need to look at the entire previous row for every cell. Let's look again at the large example from above. When we're calculating the values for the second row, we're adding the minimum from the first row onto them. The only cell we can't do this for is the one that was *directly below the minimum*, as this would break the adjacency rule. For this one, it makes sense to add the second minimum.



Here's an animation of the entire algorithm.

Video:

Algorithm

The simplest way of implementing this algorithm is to base it on the animation above. This requires overwriting the input.

Implementation

```
class Solution {

    public int minCostII(int[][] costs) {

        if (costs.length == 0) return 0;
        int k = costs[0].length;
        int n = costs.length;

        for (int house = 1; house < n; house++) {

            // Find the minimum and second minimum color in the PREVIOUS row.
            int minColor = -1; int secondMinColor = -1;
            for (int color = 0; color < k; color++) {
                int cost = costs[house - 1][color];
                if (minColor == -1 || cost < costs[house - 1][minColor]) {
                    secondMinColor = minColor;
                    minColor = color;
                } else if (secondMinColor == -1 || cost < costs[house - 1][secondMinColor]) {
                    secondMinColor = color;
                }
            }

            // And now calculate the new costs for the current row.
            for (int color = 0; color < k; color++) {
                if (color == minColor) {
                    costs[house][color] += costs[house - 1][secondMinColor];
                } else {
                    costs[house][color] += costs[house - 1][minColor];
                }
            }
        }
```

```
        // Find the minimum in the last row.
        int min = Integer.MAX_VALUE;
        for (int c : costs[n - 1]) {
            min = Math.min(min, c);
        }
        return min;
    }
}
```

Complexity Analysis

- Time complexity : $O(n \cdot k)$.

  The first loop that finds the minimums of the first row is $O(k)$ because it looks at each of the k values in the first row exactly once. The second loop is $O(n \cdot k)$ because the outer loop loops n times, and the inner loop loops k times. $O(n \cdot k) + O(k) = O(n \cdot k)$. We know it is *impossible* to ever do better here, because we cannot solve the problem without at least looking at each of the $n \cdot k$ cells once.

- Space complexity : $O(1)$.

  Like approach 2, this approach also modifies the input instead of allocating its own space.

Approach 5: Dynamic programming with Optimized Time and Space

Intuition

There is another way we can still solve the problem in $O(1)$ space and $O(n \cdot k)$ time complexity, *and* preserving the input.

The only thing the algorithm in the previous approach is really doing is going through the rows, and finding the 2 minimums of each row. It does this by calculating all the new costs for the row, writing them into the input, and then finding the minimums. This overwriting isn't necessary though—we can simply keep track of the 2 smallest values we've seen so far, as we go, in the current row. We also need to remember the 2 from the previous row.

Algorithm

The approach is a hybrid of approach 3 and 4. Like approach 4, it finds the minimums once instead of repeatedly. Like approach 3, it keeps track of information only from the current and previous rows. Unlike approach 3 though, the only information kept is the minimums.

Implementation

```
class Solution {

    public int minCostII(int[][] costs) {
```

```
if (costs.length == 0) return 0;
int k = costs[0].length;
int n = costs.length;



/* Firstly, we need to determine the 2 lowest costs of
 * the first row. We also need to remember the color of
 * the lowest. */
int prevMin = -1; int prevSecondMin = -1; int prevMinColor = -1;
for (int color = 0; color < k; color++) {
    int cost = costs[0][color];
    if (prevMin == -1 || cost < prevMin) {
        prevSecondMin = prevMin;
        prevMinColor = color;
        prevMin = cost;
    } else if (prevSecondMin == -1 || cost < prevSecondMin) {
        prevSecondMin = cost;
    }
}

// And now, we need to work our way down, keeping track of the minimums.
for (int house = 1; house < n; house++) {
    int min = -1; int secondMin = -1; int minColor = -1;
    for (int color = 0; color < k; color++) {
        // Determine the cost for this cell (without writing it in).
        int cost = costs[house][color];
        if (color == prevMinColor) {
            cost += prevSecondMin;
        } else {
            cost += prevMin;
```

```
                }
                // Determine whether or not this current cost is also a minimum.
                if (min == -1 || cost < min) {
                    secondMin = min;
                    minColor = color;
                    min = cost;
                } else if (secondMin == -1 || cost < secondMin) {
                    secondMin = cost;
                }
            }
            // Transfer current mins to be previous mins.
            prevMin = min;
            prevSecondMin = secondMin;
            prevMinColor = minColor;
        }

        return prevMin;
    }
}
```

There are many ways to compact the code a bit more, particularly in the case of the Python. I haven't done this here as it could be problematic for those less familiar with the 2 languages I have provided solutions in, however feel free to post your own solutions in the comments. I'm excited to see the elegance you can come up with!

Complexity Analysis

- Time complexity : $O(n \cdot k)$.

  Same as the previous approach.

- Space complexity : $O(1)$.

  The only additional working memory we're using is a constant number of single-value variables to keep track of the 2 minimums in the current and previous row, and to calculate the cost of the current cell. Because the memory usage is constant, we say it is $O(1)$. Unlike the previous approach one though, this one does not overwrite the input.