256. Paint House

Difficulty	Medium
Tag	Dynamic Programming
Vote of Good or Bad	Good

There is a row of `n` houses, where each house can be painted one of three colors: red, blue, or green. 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 3` cost matrix `costs`.

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

Return *the minimum cost to paint all houses*.

Example 1:

```
Input: costs = [[17,2,17],[16,16,5],[14,3,19]]
Output: 10
Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2 into blue.
Minimum cost: 2 + 5 + 3 = 10.
```

Example 2:

```
Input: costs = [[7,6,2]]
Output: 2
```

Constraints:

- `costs.length == n`

- `costs[i].length == 3`

- `1 <= n <= 100`

- `1 <= costs[i][j] <= 20`

Solution

For those already familiar with memoization and dynamic programming, this question will be easy. For those who are very new to Leetcoding, it might seem like a medium, or even a hard. ***For those who are starting to learn about memoization and dynamic programming, this question is a great one for getting started***!

This article is aimed at those of you getting started with dynamic programming and memoization. I'll assume you have already worked through prerequisite concepts such as n-ary trees (or binary trees), including with recursion. If you haven't, then I strongly recommend that you come back to this question after working through either the N-ary Trees module or the Binary Trees module. The intuition behind memoization and dynamic programming is best understood using trees, so that is what I've done in this article. Understanding how to recognize and then approach memoization and dynamic programming problems is essential for interview success.

Approach 1: Brute force

Intuition

The brute force approach is often a good place to start. From there, we can identify unnecessary work and further optimize. In this case, the brute force algorithm would be to generate every valid permutation of house colors (or all permutations and then remove all the invalid ones, e.g. ones that have 2 red houses side-by-side) and score them. Then, the lowest score is the value we need to return.

For this article, we'll use the following input. It is for 4 houses.

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

Color

House	Red (0)	Green (1)	Blue (2)
0	17	2	17
1	8	4	10
2	6	3	19
3	4	8	12

These are all the valid sequences you can get with 4 houses. In total, there are 24 of them. The one with the lowest total cost is highlighted.

The best option is to paint the first house green, second house red, third house green, and fourth house red. This will cost a total of 17 .

Algorithm

It's not worth worrying about how you'd implement the brute force solution—it's completely infeasible and useless in practice. Additionally, the latter approaches move in a different direction, and the permutation code actually takes some effort to understand (which would be a distraction for you). Therefore, I haven't included code for it. You wouldn't be writing code for it in an interview either, instead you'd simply describe a possible approach and move onto optimizing, and then write code for a more optimal algorithm.

There are many different approaches to it. All are based on permutation generation, but some only generate permutations that follow the color rules, and others generate all permutations and then remove the non-valid ones afterwards. Some are recursive, and others are iterative. Some use $O(n)$ space by only generating one permutation at a time and then processing it before generating the next, and others use a lot more (discussed below) from generating all the sequences first and then processing them.

The simplest is probably to generate every possible length-n string of `0`, `1`, and `2`, remove any that have the same digit twice in a row, and then score those that are left, keeping track of the smallest cost seen so far.

Complexity Analysis

- Time complexity : $O(2\text{\textasciicircum}n)$ $O(2n)$ or $O(3\text{\textasciicircum}n)$ $O(3n)$.

  Without writing code, we can get a good idea of the cost. We know that at the very least, we'd have to process every valid permutation. The number of valid permutations doubles with every house added. With `4` houses, there were `24` permutations. If we add another house, then all of our permutations for 4 houses could be extended with 2 different colors for the 5th house, giving `48` permutations. Because it doubles every time, this is $O(n\text{\textasciicircum}2)$ $O(n2)$.

  It'd be even worse if we generated all permutations of `0`, `1`, and `2` and then pruned out the invalid ones. There are $O(n\text{\textasciicircum}3)$ $O(n3)$ such permutations in total.

- Space complexity : Anywhere from $O(n)$ $O(n)$ to $O(n \cdot 3\text{\textasciicircum}n)$ $O(n \cdot 3n)$.

  This would depend entirely on the implementation. If you generated all the permutations at the same time and put them in a massive list, then you'd be using $O(n * 2\text{\textasciicircum}n)$ $O(n*2n)$ or $O(n * 3\text{\textasciicircum}n)$ $O(n*3n)$ space. If you generated one, processed it, generated the next, processed it, etc, without keeping the long list, it'd require $O(n)$ $O(n)$ space.

## Approach 2: Brute force with a Recursive Tree

Intuition

Like the first approach, this approach still isn't good enough. However, it bridges the gap between approach 1 and approach 3, with approach 3 further building on it. So make sure you understand it well.

When we have permutations, we can think of them as forming a big tree of all the options. Drawing out the tree (or part of it) can give useful insights and reveal other possible algorithms. We'll continue using the sample example that we did above:

Color

| House	Red (0)	Green (1)	Blue (2)
0	17	2	17
1	8	4	10
2	6	3	19
3	4	8	12

And here is how we can represent it using a tree. Each path from root to leaf represents a different possible permutation of house colors. There are 24 leaf nodes on the tree, just like there was 24 permutations identified in the brute force approach.
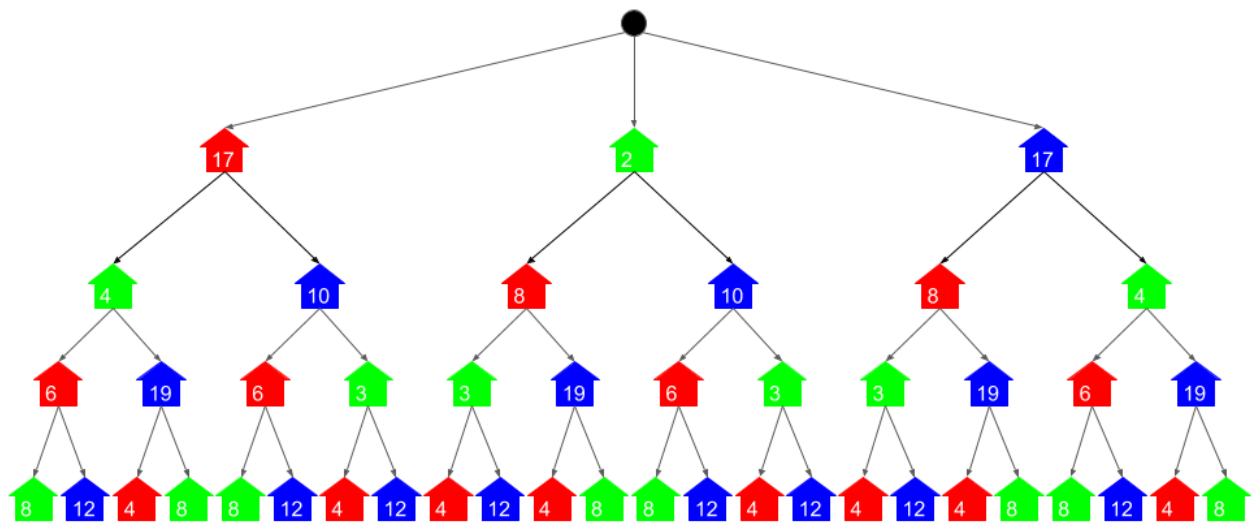
The tree representation gives a useful model of the problem and all the possible permutations. It shows that, for example, if we paint the first house red, then we have 2 options for the second house: green or blue. And then if we choose green for the second house, we could choose red or blue for the 3rd house. And so forth.

Without worrying yet about how we would actually implement it, we'll now explore a straightforward algorithm that can be used to solve the problem using this tree.

If the first 3 houses were red, green, and red then we could paint the 4th house green or blue. Which would we want to choose?

Which of these 2 would we pick?
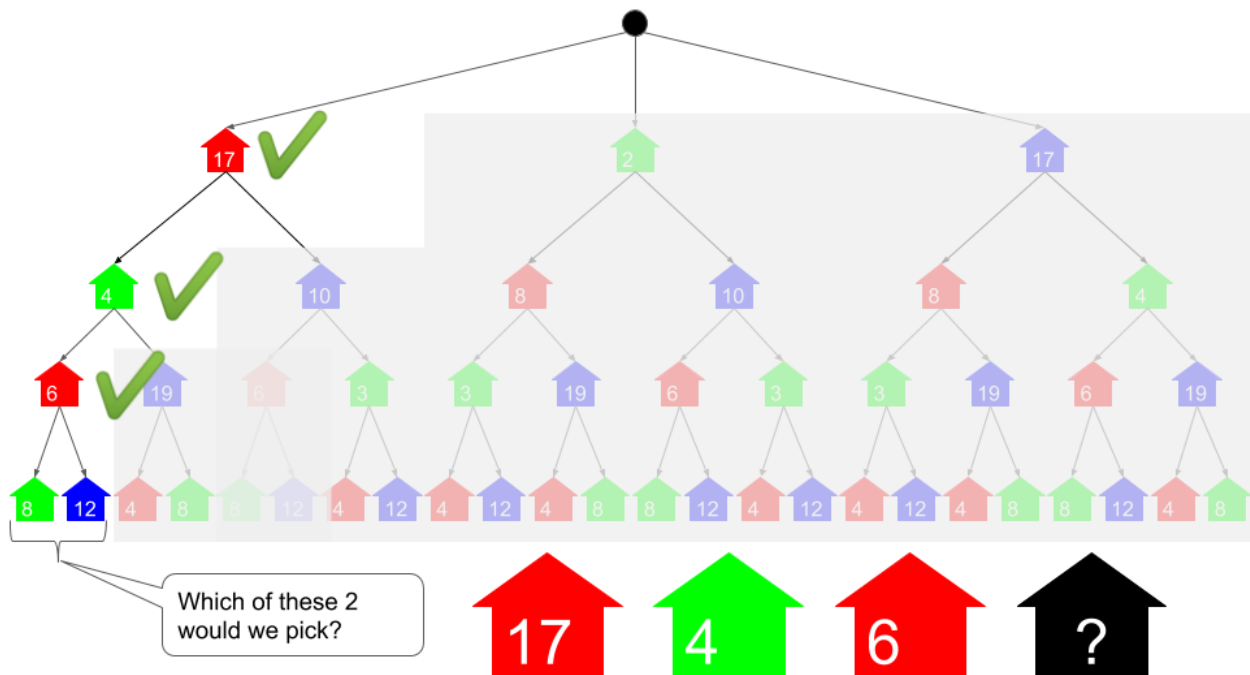
To minimize cost, we'd choose green. This is because green is `8`, and blue is `12`. *Under the assumption that we'd already decided that the first 3 houses would be red, green, and red*, this decision is definitely optimal. We know that there's no way we could do better.

What we were effectively doing was deciding which was cheaper out of 2 permutations: `red, green, red, green` or `red, green, red, blue`. Because the former is cheaper, we have completely ruled out the latter. We can simplify our tree with this new information by adding the cost of the 4th house to the cost of the 3rd house on that branch.

Becomes 6 + 8 = 14
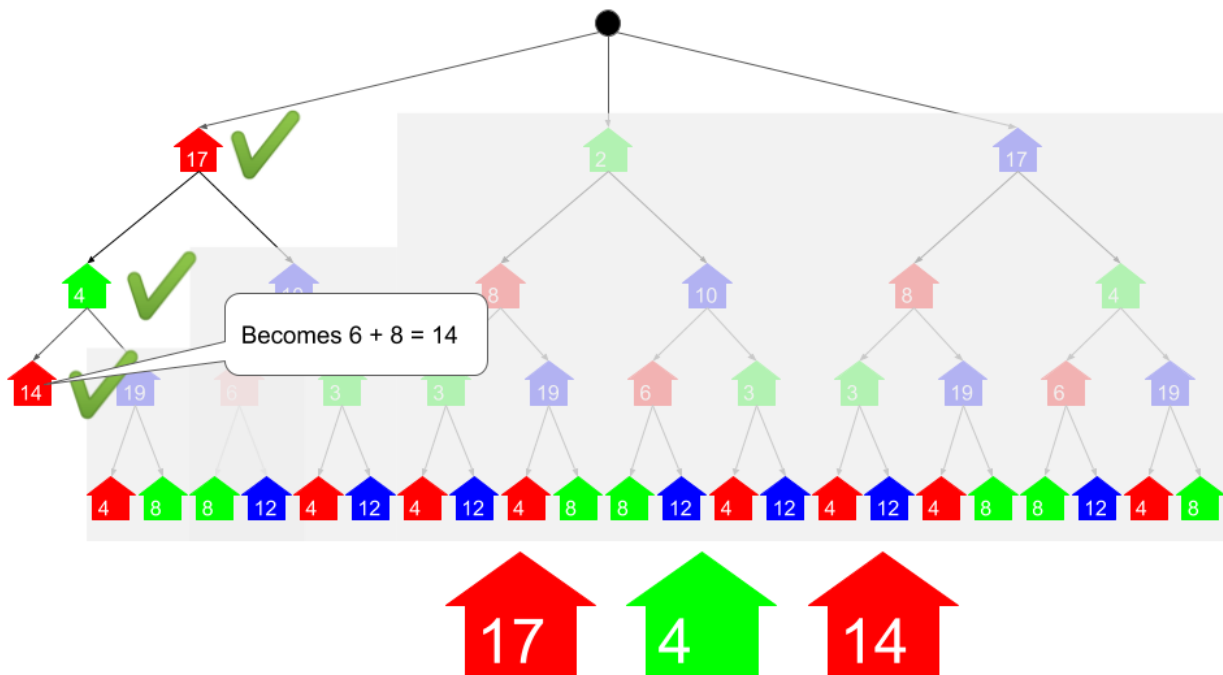
We can repeatedly remove leaf nodes following this same process, as shown in this animation.

1 / 27

We are left with the conclusion that:

- Painting the first house red would have a *total* cost of `34` .
- Painting the first house green would have a *total* cost of `17` .
- Painting the first house blue would have a *total* cost of `32` .

So, it makes sense to paint the first house green. This gives a total cost of `17` , which was the same answer our brute force in approach 1 arrived at.

Algorithm

To actually implement it, we'll need to change the way we think about it. What we did here was a *bottom-up* algorithm, meaning that we started by processing leaf nodes and then worked our way up. When we implement algorithms like this though, we almost always do it *top-down*. This allows us to use an *implicit tree* with recursion, instead of actually making a tree (i.e. having to work with `TreeNode` 's'). The recursive calls all form a tree structure. If you're not too

familiar with this idea yet, don't panic, there is an animation of the algorithm and the code in the next section. The best way to get your head around recursion is to look at examples and recognise common patterns.

Let's get started. Remember how we determined the cost of painting each house in the tree?

The total cost of painting the first house red is the cost of the red paint for the first house, AND the minimum total cost out of the green and blue house. Just like before.
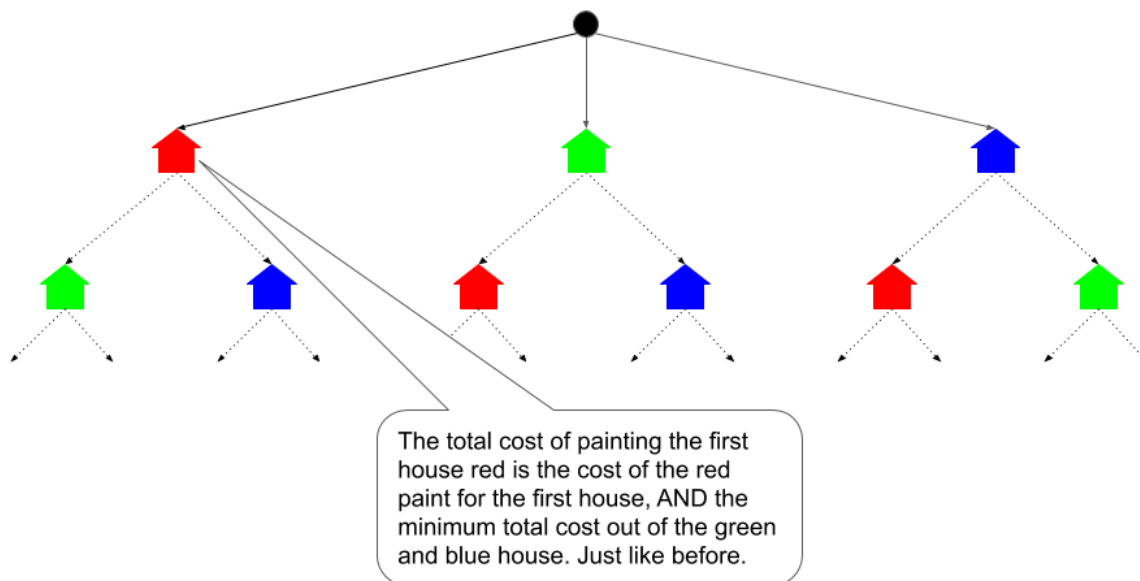
By *total cost*, we mean the cost of painting that house a particular color *and* painting the ones after it optimally.

In pseudocode the top-down recursive algorithm looks like this:

```
print min(paint(0, 0), paint(0, 1), paint(0, 2))

define function paint(n, color):
  total_cost = costs[n][color]
  if n is the last house number:
    pass [go straight to the return]
  else if color is red (0):
    total_cost += min(paint(n+1, 1), paint(n+1, 2))
  else if color is green (1):
    total_cost += min(paint(n+1, 0), paint(n+1, 2))
  else if color is blue (2):
    total_cost += min(paint(n+1, 0), paint(n+1, 1))
  return the total_cost
```

Here is an animation/ walkthrough of the algorithm. It also shows how the recursive calls make the same structure as the tree we were playing around with before, without actually building a tree. While this algorithm might be a bit to get your head around if you're not too familiar with recursion, doing so is essential to understanding approach 3.

1 / 34

And here is the code. While you're reading over it, have an initial think about how you could optimize it so that it no longer takes exponential time. Hint: look closely at the parameters of the recursive function. Are we actually repeating the same thing over and over? Fixing this problem will be what we tackle in Approach 3.
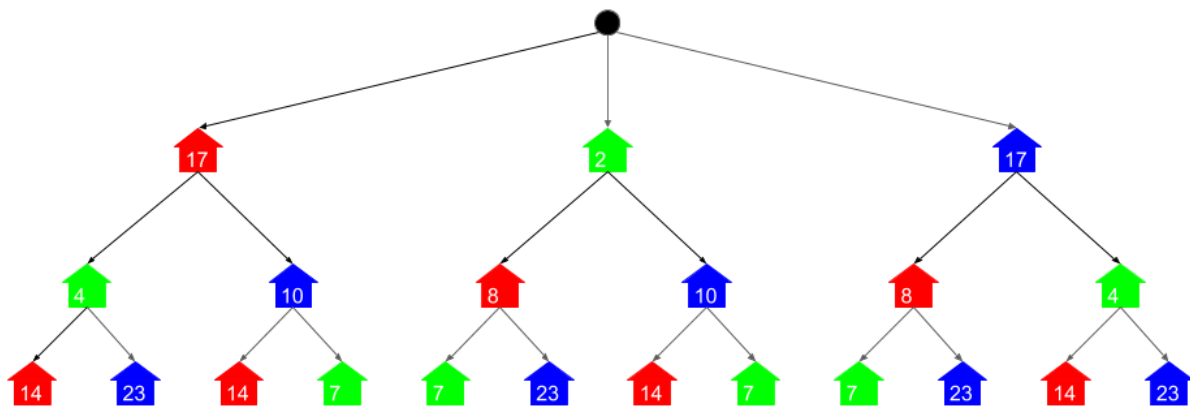
Complexity Analysis

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

  While this approach is an improvement on the previous approach, it still requires exponential time. Think about the number of leaf nodes. Each permutation has its own leaf node. The number of internal nodes is the same as the number of leaf nodes too. Remember how there are 2^n$2n$ different permutations? Each effectively adds `2` nodes to the tree, so dropping the constant of `2` gives us O(2^n)$O(2n)$.

  This is better than the previous approach, which had an additional factor of `n`, giving O(n \cdot 2 ^ n)$O(n·2n)$. That extra factor of `n` has disappeared here because the permutations are now "sharing" their similar parts, unlike before. The idea of "sharing" similar parts can be taken much further for this particular problem, as we will see with the remaining approaches that knock the time complexity all the way down to O(n)$O(n)$.

- Space complexity : O(n)$O(n)$.

  This algorithm might initially appear to be O(1)$O(1)$, because we are not allocating any new data structures. However, we need to take into account space usage on the **run-time stack**. The run-time stack was shown in the animation. Whenever we are processing the last house (house number `n - 1`), there are `n` stack frames on the stack. This space usages counts for

complexity analysis (it's memory usage, like any other memory usage) and so the space complexity is O(n)$O(n)$.

Approach 3: Memoization

Intuition

You may have noticed a very important pattern while we were working on the previous approach. Let's take a closer look.

This is the tree before we removed any layers.

Look at the leaf nodes. All the red houses cost 4 , the green houses 7 , and the blue houses 23 . This makes sense, as the original input told us the costs of painting the 4th house red, green, or blue were 4 , 7 , and 23 respectively.

But look at what happens when we remove those leaf nodes in the way we described in the previous section.

Again, all the red houses are the same at 14 , the green houses are 7 , and the blue houses are 23 . Why has this happened? Well, we were always adding the cheapest of the 2 children to the parent, before deleting the 2 children. Painting the 3rd house itself red *always* costs 6 . And then we can *always* choose between painting the 4th house green or blue. It only ever made sense to choose green, as that was 8 (compared to 12 to paint it blue) Therefore, all those branches became $6 + 8 = 14$. Similar arguments apply to painting the 3rd house blue or green.

And here's the tree when we'd removed another layer again.
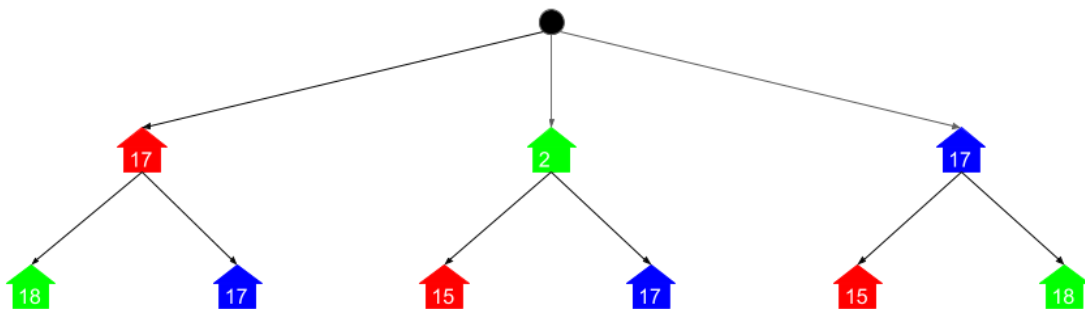
Unsurprisingly, the pattern still continues.

This pattern is important, because it shows us that we're actually doing the same few calculations over and over again. Instead of repeatedly doing the same (expensive) calculations, we should instead save and re-use results where possible.

For example, imagine if in school you'd been given this math homework (and were *not* allowed to use a calculator). How would you approach it?

```
1) 345 * 282 = ?
2) 43 + (345 * 282) = ?
3) (345 * 282) + 89 = ?
4) (345 * 282) * 5 + 19 = ?
```

Unless you really, really love arithmetic, I think you would have done the working for `345 * 282` just *once* and then inserted it into all the other equations. You probably wouldn't have done the long multiplication 4 separate times for it!

And it's the same for calculating the costs for painting these houses. We only need to calculate the cost of painting the 2nd house red *once*.

So to do this, we'll use **memoization**. Immediately before returning a value we've finished computing, we'll write it into a dictionary with the input values as the key and the return value as the result. Then at the start of the function, we'll first check if the answer is already in the dictionary. If it is, we can immediately return it. If not, then we need to continue like before and compute it.

Algorithm

The algorithm is almost the same as before. The only difference is that we create an empty dictionary at the start, write the return values into it, and check it first to see if we've already found the answer for a particular set of input parameters.

```
print min(paint(0, 0), paint(0, 1), paint(0, 2))

memo = a new, empty dictionary

define function paint(n, color):
  if (n, color) is a key in memo:
     return memo[(n, color)]
  total_cost = costs[n][color]
  if n is the last house number:
    pass [go straight to return]
  else if color is red (0):
    total_cost += min(paint(n+1, 1), paint(n+1, 2))
  else if color is green (1):
    total_cost += min(paint(n+1, 0), paint(n+1, 2))
  else if color is blue (2):
    total_cost += min(paint(n+1, 0), paint(n+1, 1))
```

```
    memo[(n, color)] = total_cost
    return the total_cost
```

Remember how the previous approach made a recursive function call for every node in the tree we drew? Well this approach only needs to do the calculations shown. The brighter circles represent where it needed to actually calculate the answer and the dull circles show where an answer was looked up in the dictionary.
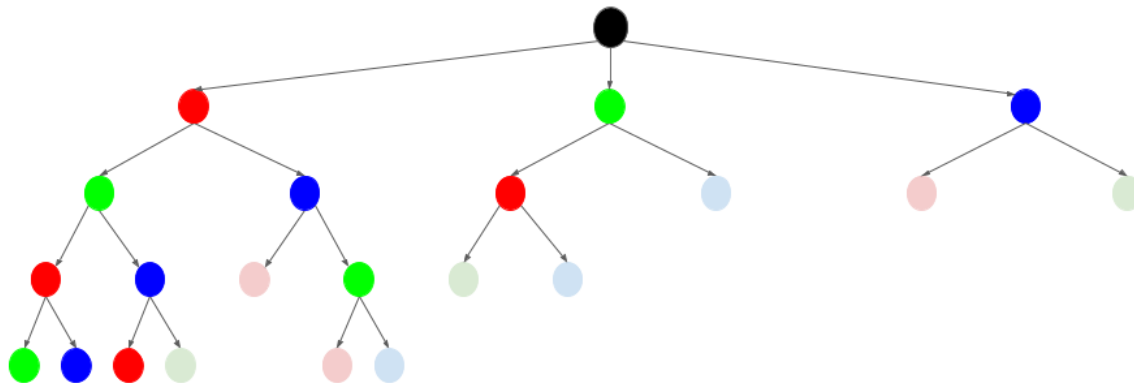
In Python, we can use the `lru_cache` decorator from the `functools` package. If you're not familar with this, you can find it <u>In the python documentation</u>. It's very useful!

Here's the code using it.

Complexity Analysis

- Time complexity : O(n)$O(n)$.

  Analyzing memoization algorithms can be tricky at first, and requires understanding how recursion impacts the cost differently to loops. The key thing to notice is that the full function runs once for each possible set of parameters. There are `3 * n` different possible sets of parameters, because there are `n` houses and `3` colors. Because the function body is O(1)$O(1)$ (it's simply a conditional), this gives us a total of `3 * n`. There can't be more than `3 * 2 * n` searches into the memoization dictionary either. The tree showed this clearly—the nodes representing lookups had to be the child of a call where a full calculation was done. Because the constants are all dropped, this leaves O(n)$O(n)$.

- Space complexity : O(n)$O(n)$.

  Like the previous approach, the main space usage is on the stack. When we go down the first branch of function calls (see the tree visualization), we won't find any results in the dictionary. Therefore, every house will make a stack frame. Because there are n houses, this gives a worst case space usage of O(n)$O(n)$. Note that this could be a problem in languages such as Python, where stack frames are large.

Approach 4: Dynamic Programming

Intuition

In approach 2, we started with, although didn't actually implement, a bottom up algorithm. The reason we didn't implement it is because we would have had to generate an actual tree which would have been a lot of work, and unnecessary for what we were trying to accomplish. However, there is another way of writing an iterative bottom-up algorithm to solve this problem. It utilizes the same pattern that we identified in approach 3.

As a starting point, what would the tree look like if we converted it into a directed graph without the repetition? In other words, if we made it so that the 2nd house being blue was pointed to by both the 1st house being green and the 1st house being red? Well, it'd look something like this.
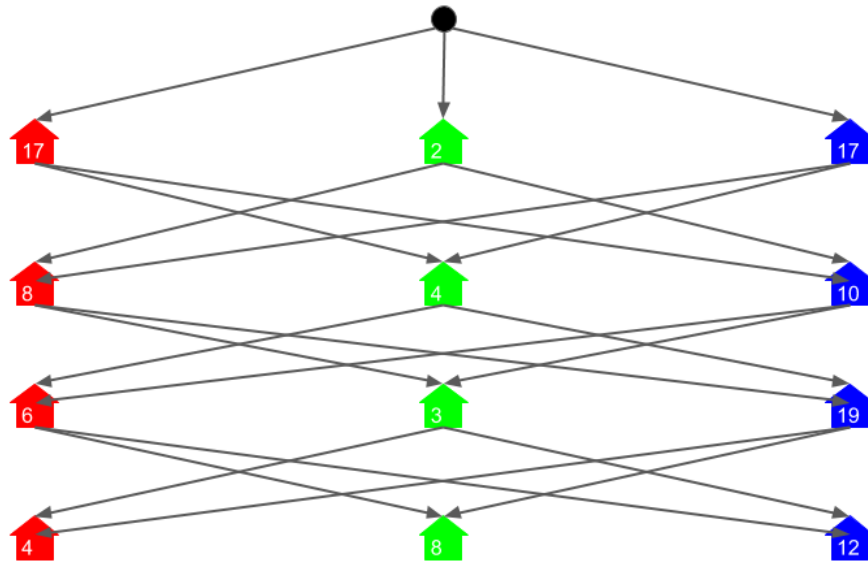
Directly generating this graph (i.e. not generating the massive tree first) and then using the same algorithm from approach 2 would achieve comparable time complexity to approach 3. But there's a far simpler way that doesn't even require generating the graph: dynamic programming! Dynamic programming is iterative, unlike memoization, which is recursive.

We'll define a subproblem to be calculating the total cost for a particular house position and color.

For the 4-house example, the memoization approach needed to solve a total of 12 different subproblems. We know this, because there were 3 possible values for the color (0, 1, 2), and 4 possible values for the house number (0, 1, 2, 3). In total, this gave us 12 different possibilities. The dynamic programming approach will need to solve these same subproblems, except in an iterative manner.

Color

	Red (0)	Green (1)	Blue (2)
0	total_cost(0, 0)	total_cost(0, 1)	total_cost(0, 2)
1	total_cost(1, 0)	total_cost(1, 1)	total_cost(1, 2)
2	total_cost(2, 0)	total_cost(2, 1)	total_cost(2, 2)
3	total_cost(3, 0)	total_cost(3, 1)	total_cost(3, 2)

House

Now, remember the size of the input array? It's the same! Also, notice how it maps onto the tree. Again, it's the same.

Color

	Red (0)	Green (1)	Blue (2)
0	17	2	17
1	8	4	10
2	6	3	19
3	4	8	12

House

We can, therefore, calculate the cost of each subproblem, starting from the ones with the highest house numbers, and write the results directly into the input array. In effect, we will replace each single-house cost value in the array with the cost of painting the house that color and the minimum cost to paint all the houses after it. This is almost the same as what we did on the tree. The only difference is that we are only doing each calculation once and we are writing results directly into the input table. It is bottom up, because we are solving the "lower" problems first, and then the "higher" ones once we've solved all the lower ones that they depend on.
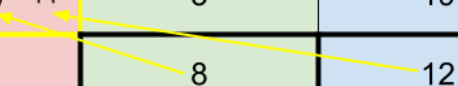
First thing to realize is that we don't need to do anything to the last row. Like in the tree, these costs are the total costs because there are no further houses after them.

	Red (0)	Green (1)	Blue (2)	
0	17	2	17	
1	8	4	10	
2	6	3	19	
3	4	8	12	✔️

Now, what about the second-to-last row? Well, we know that if we painted that house red, that it'd cost itself and the cheapest out of blue and green from the next row, which is 8. So the total cost there would be 14, and we can put that into the cell.

	Red (0)	Green (1)	Blue (2)
0	17	2	17
1	8	4	10
2	6 + min(8, 12) = 14	3	19
3	4	8	12

Just like we did with the tree, we can work our way up through the grid, repeatedly applying the same algorithm to determine the total value for each cell. Once we have updated all the cells, we then simply need to take the minimum value from the first row and return it. Here is an animation showing the process.

Algorithm

The algorithm is straightforward. We iterate backwards over all the rows in the grid (starting from the second-to-last) and calculate a total cost for each cell in the way shown in the animation.

You could also avoid the hardcoding of the colors and instead iterate over the colors. This approach will be covered in the solution article for the follow up question where there are m colors instead of just 3.

Complexity Analysis

- Time Complexity : O(n)$O(n)$.

   Finding the minimum of two values and adding it to another value is an O(1)$O(1)$ operation. We are doing these O(1)$O(1)$ operations for 3 \cdot (n - 1)$3 \cdot (n-1)$ cells in the grid. Expanding that out, we get 3 \cdot n - 3$3 \cdot n - 3$. The constants don't matter in big-oh notation, so we drop them, leaving us with O(n)$O(n)$.

   A word of warning: This would *not* be correct if there were mm colors. For this particular problem we were told there's only 33 colors. However, a logical follow-up question would be to make the code work for any number of colors. In that case, the time complexity would actually be O(n \cdot m)$O(n \cdot m)$, because mm is not a constant, whereas 33 is. If this confused you, I'd recommend reading up on big-oh notation.

- Space Complexity : O(1)$O(1)$

   We don't allocate any new data structures, and are only using a few local variables. All the work is done directly into the input array. Therefore, the algorithm is in-place, requiring constant extra space.

Approach 5: Dynamic Programming with Optimized Space Complexity

Intuition

Overwriting the input array isn't always desirable. What if, for example, other functions also needed to use that same array?

We could allocate our own array and then continue in the same way as approach 4. This would bring our space complexity up to O(n)$O(n)$ (for the same reason the time complexity is O(n)$O(n)$, the constants are dropped in big-oh notation).

Using O(n)$O(n)$ space isn't necessary though—we can further optimize the space complexity. Remember how the dynamic programming animation blanked out rows to show we'd no longer be looking at them? We only needed to look at the previous row, and the row we're currently working on. The rest could have been thrown away. So to avoid overwriting the input, we keep track of the previous row and the current row as length-3 arrays.

This space-optimization technique applies to many dynamic programming problems. As a general rule, I'd recommend first trying to come up with an algorithm that has optimal time complexity, and then looking at if you can trim down the space complexity.

Algorithm

It's up to you whether you do this using length-3 arrays or variables. Arrays are better in terms of writing clean code though. They will also be easier to adapt if you were asked to make the algorithm work with mm colors. I have chosen to use arrays here as keeping track of 6 seperate variables is too messy.

The `previous_row` starts as being the last row of the input array. The `current_row` is the row `n` is currently up to (starts as the second to last row). At each step we update the values in `current_row` by adding values from `previous_row`. We then set `previous_row` to be `current_row` and go on to the next value of `n` where we repeat the process. At the end, the first row will be sitting in the `previous_row` variable, so we find the minimum like we did before.

Note that we have to be careful about not overwriting the `costs` array inadvertently. Any rows we take out of the array that will be *written* into will need to be copies. This can be done using `clone` in Java (suitable for an array of primitive types such as integers) and `copy.deepcopy` in Python.

Thanks so much to @bitbleach for pointing out that the original code I had here was over writing the input array! Because this is such an easy mistake to make, I've kept the original code for reference.

Complexity Analysis

- Time Complexity : O(n)$O(n)$.

  Same as previous approach.

- Space Complexity : O(1)$O(1)$

  We're "remembering" up to 66 calculations at a time (using 2 x length-3 arrays). Because this is actually a constant, the space complexity is still O(1)$O(1)$.

  Like the time complexity though, this analysis is dependent on there being a constant number of colors (i.e. 3). If the problem was changed to be mm colors, then the space complexity would become O(m)$O(m)$ as we'd need to keep track of a couple of length-m arrays.

Justifying why this is a Dynamic Programming Problem

Many dynamic programming problems have very straightforward solutions. As you get more experience with them, you'll gain a better intuition for when a problem might be solvable with dynamic programming, and you'll also get better at quickly identifying the overlapping subproblems (e.g. that painting the 3rd house green will have the same total cost regardless of whether the 2nd house was blue or red). Thinking about the tree structure can help too for identifying those subproblems, although you won't always need to draw it out fully like we did here.

Remember that a **subproblem** is any call to the recursive function. Subproblems are solved either as a base case (in this case a simple lookup from the table and no further calculations) or by looking at the solutions of a bunch of lower down subproblems. In dynamic programming lingo, we say that this problem has an **optimal substructure**. This means that the optimal cost for each **subproblem** is constructed from the **optimal cost** of **subproblems** below it. This is the same property that must be true for greedy algorithms to work.

If, for example, we hadn't been able to choose the minimum and know it was optimal (perhaps because it would impact a choice further up the tree) then there would *not* have been **optimal substructure**.

In addition this problem also had **overlapping subproblems**. This just means that the lower subproblems were often shared (remember how the tree had lots of branches that looked the same?)

Problems that have **optimal substructure** can be solved with greedy algorithms. If they *also* have **overlapping subproblems**, then they can be solved with dynamic programming algorithms.