# Paint Fence

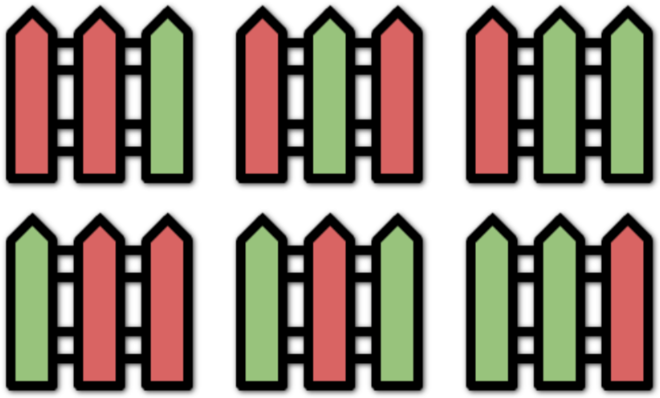
Solution ★★★★★

You are painting a fence of `n` posts with `k` different colors. You must paint the posts following these rules:

- Every post must be painted **exactly one** color.
- There **cannot** be three or more **consecutive** posts with the same color.

Given the two integers `n` and `k`, return *the **number of ways** you can paint the fence*.

**Example 1:**



```
Input: n = 3, k = 2
Output: 6
Explanation: All the possibilities are shown.
Note that painting all the posts red or all the posts green is invalid because there cannot be three posts in a r
```

**Example 2:**

```
Input: n = 1, k = 1
Output: 1
```

**Example 3:**

```
Input: n = 7, k = 2
Output: 42
```

**Constraints:**

- `1 <= n <= 50`
- `1 <= k <=` $10^5$
- The testcases are generated such that the answer is in the range `[0,` $2^{31}$ `- 1]` for the given `n` and `k`.

C++

```
class Solution {
public:
    int numWays(int n, int k) {

    }
};
```

# Solution

## Overview

**Realizing This is a Dynamic Programming Problem**

There are two parts to this problem that tell us it can be solved with dynamic programming.

First, the question is asking for the "number of ways" to do something.

Second, we need to make decisions that may depend on previously made decisions. In this problem, we need to decide what color we should paint a given post, which may change depending on previous decisions. For example, if we paint the first two posts the same color, then we are not allowed to paint the third post the same color.

Both of these things are characteristic of dynamic programming problems.

**A Framework to Solve Dynamic Programming Problems**

A dynamic programming algorithm typically has 3 components. Learning these components is extremely valuable, as **most dynamic programming problems can be solved this way**.

First, we need some function or array that represents the answer to the problem for a given state. For this problem, let's say that we have a function `totalWays`, where `totalWays(i)` returns the number of ways to paint `i` posts. Because we only have one argument, this is a one-dimensional dynamic programming problem.

Second, we need a way to transition between states, such as `totalWays(3)` and `totalWays(4)`. This is called a **recurrence relation** and figuring it out is usually the hardest part of solving a problem with dynamic programming. We'll talk about the recurrence relation for this problem below.

The third component is establishing base cases. If we have one post, there are `k` ways to paint it. If we have two posts, then there are `k * k` ways to paint it (since we are allowed to paint have two posts in a row be the same color). Therefore, `totalWays(1) = k, totalWays(2) = k * k`.

**Finding The Recurrence Relation**

We know the values for `totalWays(1)` and `totalWays(2)`, now we need a formula for `totalWays(i)`, where `3 <= i <= n`. Let's think about how many ways there are to paint the $i^{th}$ post. We have two options:

1. Use a different color than the previous post. If we use a different color, then there are `k - 1` colors for us to use. This means there are `(k - 1) * totalWays(i - 1)` ways to paint the $i^{th}$ post a different color than the $(i-1)^{th}$ post.

2. Use the same color as the previous post. There is only one color for us to use, so there are `1 * totalWays(i - 1)` ways to paint the $i^{th}$ post the same color as the $(i-1)^{th}$ post. However, we have the added restriction of not being allowed to paint three posts in a row the same color. Therefore, we can paint the $i^{th}$ post the same color as the $(i-1)^{th}$ post **only if** the $(i-1)^{th}$ post is a different color than the $(i-2)^{th}$ post.

   So, how many ways are there to paint the $(i-1)^{th}$ post a different color than the $(i-2)^{th}$ post? Well, as stated in the first option, there are `(k - 1) * totalWays(i - 1)` ways to paint the $i^{th}$ post a different color than the $(i-1)^{th}$ post, so that means there are `1 * (k - 1) * totalWays(i - 2)` ways to paint the $(i-1)^{th}$ post a different color than the $(i-2)^{th}$ post.

Adding these two scenarios together gives `totalWays(i) = (k - 1) * totalWays(i - 1) + (k - 1) * totalWays(i - 2)`, which can be simplified to:

`totalWays(i) = (k - 1) * (totalWays(i - 1) + totalWays(i - 2))`

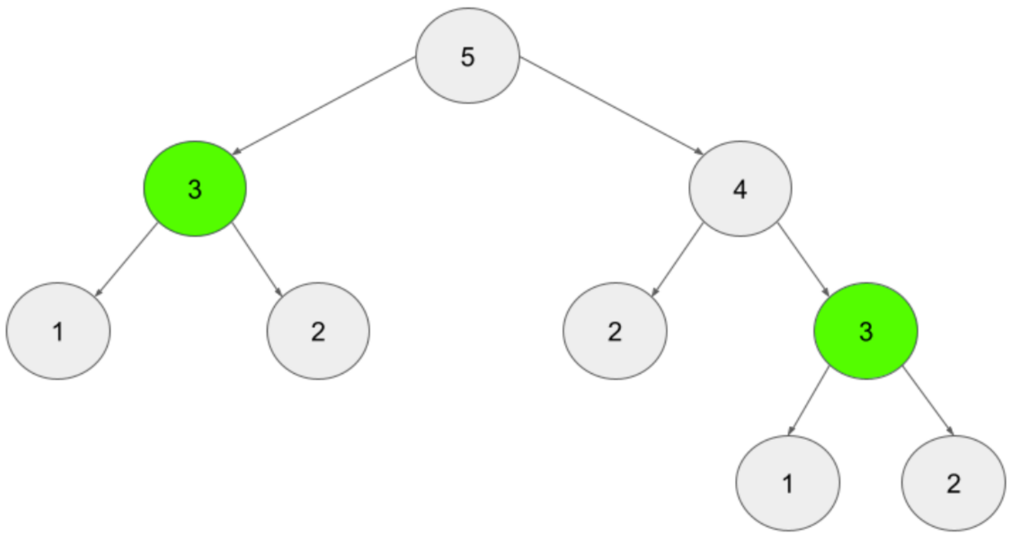
This is our recurrence relation which we can use to solve the problem from the base cases.

## Approach 1: Top-Down Dynamic Programming (Recursion + Memoization)

**Intuition**

Top-down dynamic programming starts from the top and works its way down to the base cases. Typically, this is implemented with recursion and then made efficient using *memoization*. Memoization refers to storing the results of expensive function calls to avoid duplicate computations - we'll soon see why this is important for this problem. If you're new to recursion, check out the recursion explore card.

We can implement the function `totalWays(i)` as follows - first, check for the base cases we defined above `totalWays(1) = k, totalWays(2) = k * k`. If `i >= 3`, use our recurrence relation: `totalWays(i) = (k - 1) * (totalWays(i - 1) + totalWays(i - 2))`. However, we will run into a major problem - repeated computation. If we call `totalWays(5)`, that function call will also call `totalWays(4)` and `totalWays(3)`. The `totalWays(4)` call will call `totalWays(3)` again, as illustrated below, we are calculating `totalWays(3)` twice.



We have to calculate numWays(3) two times

This may not seem like a big deal with `i = 5`, but imagine if we called `totalWays(6)`. This entire tree would be one child, and we would have to call `totalWays(4)` twice. As `n` increases, the size of the tree grows exponentially - imagine how expensive a call such as `totalWays(50)` would be. This can be solved with *memoization*. When we compute the value of a given `totalWays(i)`, let's store that value in memory. Next time we need to call `totalWays(i)`, we can refer to the value stored in memory instead of having to call the function again and going through the repeated computations.

**Algorithm**

1. Define a hash map `memo`, where `memo[i]` represents the number of ways you can paint `i` fence posts.
2. Define a function `totalWays` where `totalWays(i)` will determine the number of ways you can paint `i` fence posts.
3. In the function `totalWays`, first check for the base cases. `return k` if `i == 1`, and `return k * k` if `i == 2`. Next, check if the argument `i` has already been calculated and stored in `memo`. If so, `return memo[i]`. Otherwise, use the recurrence relation to calculate `memo[i]`, and then return `memo[i]`.
4. Simply call and return `totalWays(n)`.

**Implementation**

```
class Solution {
    private HashMap<Integer, Integer> memo = new HashMap<Integer, Integer>();

    private int totalWays(int i, int k) {
        if (i == 1) return k;
        if (i == 2) return k * k;

        // Check if we have already calculated totalWays(i)
        if (memo.containsKey(i)) {
            return memo.get(i);
        }

        // Use the recurrence relation to calculate totalWays(i)
        memo.put(i, (k - 1) * (totalWays(i - 1, k) + totalWays(i - 2, k)));
        return memo.get(i);
    }

    public int numWays(int n, int k) {
        return totalWays(n, k);
    }
}
```

**Extra Notes**

For this approach, we are using a hash map as our data structure to memoize function calls. We could also use an array since the calls to `totalWays` are very well defined (between 1 and `n`). However, a hash map is used for most top-down dynamic programming solutions, as there will often be multiple function arguments, the arguments might not be integers, or a variety of other reasons that require a hash map instead of an array. Although using an array is slightly more efficient, using a hash map here is a good practice that can be applied to other problems.

In Python, the functools module contains functions that can be used to automatically memoize a function. In LeetCode, modules are automatically imported, so you can just add the `@lru_cache(None)` wrapper to any function definition to have it automatically memoize.

```
class Solution:
    def numWays(self, n: int, k: int) -> int:
```

```
@lru_cache(None)
def total_ways(i):
    if i == 1:
        return k
    if i == 2:
        return k * k

    return (k - 1) * (total_ways(i - 1) + total_ways(i - 2))

return total_ways(n)
```

You can observe that by removing the @lru_cache(None) wrapper, on attempted submission, the code will exceed the time limit.

**Complexity Analysis**

- Time complexity: $O(n)$

  `totalWays` gets called with each index from `n` to `3`. Because of our memoization, each call will only take $O(1)$ time.

- Space complexity: $O(n)$

  The extra space used by this algorithm is the recursion call stack. For example, `totalWays(50)` will call `totalWays(49)`, which calls `totalWays(48)` etc., all the way down until the base cases at `totalWays(1)` and `totalWays(2)`. In addition, our hash map `memo` will be of size `n` at the end, since we populate it with every index from `n` to `3`.

---

## Approach 2: Bottom-Up Dynamic Programming (Tabulation)

**Intuition**

Bottom-up dynamic programming is also known as **tabulation** and is done iteratively. Instead of using a function like in top-down, let's use an array `totalWays` instead, where `totalWays[i]` represents the number of ways you can paint `i` fence posts.

As the name suggests, we now start at the bottom and work our way up to the top (`n`). Initialize the base cases `totalWays[1] = k, totalWays[2] = k * k`, and then iterate from `3` to `n`, using the recurrence relation to populate `totalWays`.

Bottom-up algorithms are generally considered superior to top-down algorithms. Typically, a top-down implementation will use more space and take longer than the equivalent bottom-up approach.

**Algorithm**

1. Define an array `totalWays` of length `n + 1`, where `totalWays[i]` represents the number of ways you can paint `i` fence posts. Initialize `totalWays[1] = k` and `totalWays[2] = k * k`.
2. Iterate from `3` to `n`, using the recurrence relation to populate `totalWays`: `totalWays[i] = (k - 1) * (totalWays[i - 1] + totalWays[i - 2])`.
3. At the end, return `totalWays[n]`.

```
class Solution {
    public int numWays(int n, int k) {
        // Base cases for the problem to avoid index out of bound issues
        if (n == 1) return k;
        if (n == 2) return k * k;

        int totalWays[] = new int[n + 1];
        totalWays[1] = k;
        totalWays[2] = k * k;
        for (int i = 3; i <= n; i++) {
            totalWays[i] = (k - 1) * (totalWays[i - 1] + totalWays[i - 2]);
        }

        return totalWays[n];
    }
}
```

**Complexity Analysis**

- Time complexity: $O(n)$

  We only iterate from `3` to `n` once, where each iteration requires $O(1)$ time.

- Space complexity: $O(n)$

  We need to use an array `totalWays`, where `totalWays.length` scales linearly with `n`.

### Approach 3: Bottom-Up, Constant Space

**Intuition**

You may have noticed that our recurrence relation from the previous two approaches only cares about 2 steps below the current step. For exampl if we are trying to calculate `totalWays[11]`, we only care about `totalWays[9]` and `totalWays[10]`. While we would have needed to calculate `totalWays[3]` through `totalWays[8]` as well, at the time of the actual calculation for `totalWays[11]`, we no longer care about any of the previous steps.

Therefore, instead of using $O(n)$ space to store an array, we can improve to $O(1)$ space by using two variables to store the results from the last two steps.

**Algorithm**

1. Initialize two variables, `twoPostsBack` and `onePostBack`, that represent the number of ways to paint the previous two posts. Since we star iteration from post three, `twoPostsBack` initially represents the number of ways to paint one post, and `onePostBack` initially represents th number of ways to paint two posts. Set their values `twoPostsBack = k, onePostBack = k * k`, because they are equivalent to our base cases..

2. Iterate `n - 2` times. At each iteration, simulate moving `i` up by one. Use the recurrence relation to calculate the number of ways for the current step and store it in a variable `curr`. "Moving up" means `twoPostsBack` will now refer to `onePostBack`, so update `twoPostsBack` `onePostBack`. `onePostBack` will now refer to the current step, so update `onePostBack = curr`.

3. In the end, return `onePostBack`, since "moving up" after the last step would mean `onePostBack` is the number of ways to paint `n` fence posts.

```
class Solution {
    public int numWays(int n, int k) {
        if (n == 1) return k;

        int twoPostsBack = k;
        int onePostBack = k * k;
        for (int i = 3; i <= n; i++) {
            int curr = (k - 1) * (onePostBack + twoPostsBack);
            twoPostsBack = onePostBack;
            onePostBack = curr;
        }

        return onePostBack;
    }
}
```

**Complexity Analysis**

- Time complexity: $O(n)$.

  We only iterate from `3` to `n` once, each time doing $O(1)$ work.

- Space complexity: $O(1)$

  The only extra space we use are a few integer variables, which are independent of input size.