
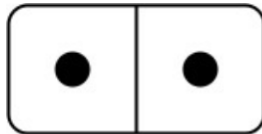


## Domino and Tromino Tiling

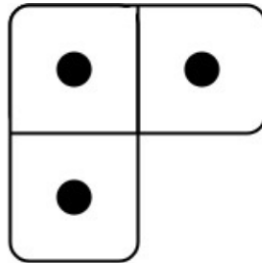
Solution 

★★★★★

You have two types of tiles: a  $2 \times 1$  domino shape and a tromino shape. You may rotate these shapes.



Domino tile

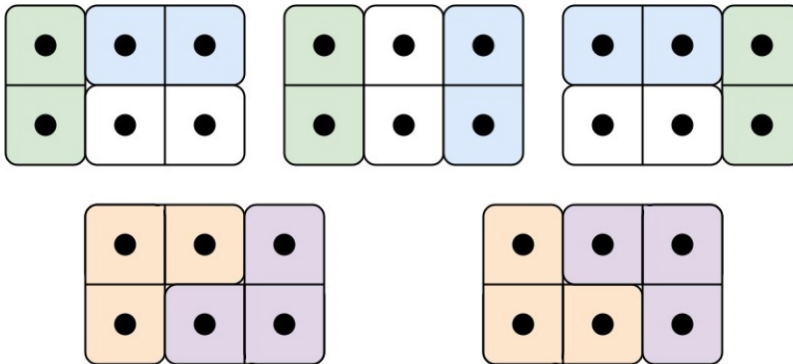


Tromino tile

Given an integer  $n$ , return the number of ways to tile an  $2 \times n$  board. Since the answer may be very large, return it modulo  $10^9 + 7$ .

In a tiling, every square must be covered by a tile. Two tilings are different if and only if there are two 4-directionally adjacent cells on the board such that exactly one of the tilings has both squares occupied by a tile.

Example 1:



Input:  $n = 3$

Output: 5

Explanation: The five different ways are shown above.

Example 2:

Input:  $n = 1$

Output: 1

### Constraints:

- $1 \leq n \leq 1000$

```
? C++  
1 class Solution {  
2 public:  
3     int numTilings(int n) {  
4  
5     }  
6 };
```

## Solution

### Overview

The objective of this question is as follows: Count how many ways to completely fill a  $2 \times n$  board using 2 shapes ( $2 \times 1$  dominos and L shaped trominos). Note that rotation of these shapes is allowed.

At first glance, one might consider testing every possible combination of dominos and trominos and then count the ones that completely fill the  $2 \times n$  board. This can be achieved through backtracking, where we fill the board from left to right and, at each step, we try all valid ways to place a domino or a tromino. Once the board is full or no tile will fit in the remaining space, we remove the last tile placed (backtrack) to return to the previous board state and try all of the remaining tile placement options. As you can imagine, constructing every possible board would be a time-consuming process. Furthermore, we are not actually interested in finding every possible way to completely fill the board - we are only interested in the number of ways the board can be completely filled. With this in mind, there should be a more efficient way to solve this problem. It seems like we are stuck now. What else can we do to solve this problem?

Whenever you are not sure how to approach a problem, it is a good idea to draw out the first couple of scenarios.

#### 1st Video:

Take a close look at the above animation. Notice that for a board with width  $k$ , some of the possible tilings can directly be derived from the two previous fully covered boards as shown below.

#### 2nd video:

#### 3rd video:

However, some of the possible tilings cannot be derived from previous fully covered boards directly.

Instead, they must be derived from partially covered boards with a width of  $k-1$  as shown below (e.g.

a fully covered board of width  $k=3$  can be derived from a partially covered board of width  $k=2$ ).

#### 4th video:

The above animations provide a basic idea of where the possible tilings come from for a board with width  $k$ . Let's find out how we can derive an algorithm from these patterns.

Now, let's define:

- **Fully covered board:** All tiles on board are covered by a *domino* or a *tromino*.
- **Partially covered board:** Same as a **fully covered board**, except leave the tile in the upper-right corner (the top row of the rightmost column) uncovered. Note, a board with only the lower-right corner uncovered is also considered "partially covered." However, as we will discover soon, we do not need to keep track of which corner is uncovered because of symmetry.
- $f(k)$ : The number of ways to **fully cover a board** of width  $k$ .
- $p(k)$ : The number of ways to **partially cover a board** of width  $k$ .

We can determine the number of ways to fully or partially tile a board of width  $k$  by considering every possible way to arrive at  $f(k)$  or  $p(k)$  by placing a domino or a tromino. Let's find  $f(k)$  together and then you can pause to practice by finding  $p(k)$  on your own. All of the ways to arrive at a fully tiled board of width  $k$  are as follows:

- From  $f(k - 1)$  we can add 1 vertical domino for each tiling in a fully covered board with a width of  $k - 1$ , as shown in the second animation.
- From  $f(k - 2)$  we can add 2 horizontal dominos for each tiling in a fully covered board with a width of  $k - 2$ , as shown in the third animation.
  - Note that we don't need to add 2 **vertical** dominos to  $f(k - 2)$ , since  $f(k - 1)$  will cover that case and it will cause duplicates if we count it again.
- From  $p(k - 1)$  we can add an L-shaped tromino for each tiling in a partially covered board with a width of  $k - 1$ , as shown above (in the fourth animation).
  - We will **multiply by  $p(k - 1)$  by 2** because for any partially covered tiling, there will be a horizontally symmetrical tiling of it. For example, the animation below shows two  $p(k - 1)$  board states that are identical when reflected over the horizontal edge of the board. Logically, there must be an equal number of ways to fully tile the board from both  $p(k - 1)$  states. So rather than count the number of ways twice, we simply multiply the number of ways from one  $p(k - 1)$  state by 2.

Summing the ways to reach  $f(k)$  gives us the following equation:

$$f(k) = f(k - 1) + f(k - 2) + 2 * p(k - 1)$$

#### 5th video:

Now that we know where tilings on  $f(k)$  are coming from, how about  $p(k)$ ? Can we apply the same logic and find that out? Absolutely yes!

Take a pen and start drawing scenarios that contribute to  $p(4)$  (this is a good technique to aid critical thinking during an interview). Start by drawing  $p(4)$ , remember  $p(4)$  is a board of width 4 with the first 3 columns fully covered and the last column half covered. Now, try removing a domino or a tromino to find which scenarios contribute to  $p(4)$ . Notice that  $p(k)$  can come from the below scenarios:

- Adding a tromino to a fully covered board of width  $k - 2$  (i.e.  $f(k - 2)$ )
- Adding a horizontal domino to a partially covered board of width  $k - 1$  (i.e.  $p(k - 1)$ )

Thus, we arrive at the following conclusion for  $p(k)$ :

$$p(k) = p(k - 1) + f(k - 2)$$

With all this information, we are very close to our first approach - **Dynamic Programming**.

#### Why Dynamic Programming?

When a question asks us to minimize, maximize, or find the number of ways to do something, it doesn't always mean that dynamic programming is the best approach, but it is usually a good indicator that we should at least consider using a dynamic programming approach.

The number of ways to reach the current state depends on the number of ways to reach the previous state. This can be seen in the functions  $f(k)$  and  $p(k)$  which depend on previous fully and partially filled boards. When using dynamic programming, these functions are called *transition functions*.

## Approach 1: Dynamic Programming (Top-down)

### Intuition

In this approach, we will use the two *transition functions* as the recurrence relation. Then we will create a recursive solution from the top ( $f(n)$ ) to the bottom (base cases described in the algorithm section) since it's generally more intuitive to solve dynamic programming problems in a top-down manner. Additionally, to avoid repeat calculations, we will memoize the result for each subproblem by storing the calculated results in two maps ( `f_cache` and `p_cache` ). Note that in the python implementation, this will be handled automatically by the `@cache` decorator.

### Algorithm

1. We'll start from  $f(n)$  and then dive all the way down to the base cases,  $f(1)$ ,  $f(2)$ , and  $p(2)$ .
2. Use the same definition for  $f$  and  $p$  from the **Overview** section
  - $f(k)$ : The number of ways to **fully cover a board** of width  $k$
  - $p(k)$ : The number of ways to **partially cover a board** of width  $k$
3. Recursion calls will use the results of subproblems and base cases to help us get the final result,  $f(n)$ .
  - The stop condition for the recursive calls is when  $k$  reaches a base case (i.e.  $k \leq 2$ ).
  - Values for the base cases will be directly returned instead of making more recursive calls.
    - $f(1) = 1$
    - $f(2) = 2$
    - $p(2) = 1$
  - To avoid repeated computations, we will use 2 hashmaps ( `f_cache` and `p_cache` ) to store calculated values for  $f$  and  $p$ . In Python, the built-in `@cache` wrapper will automatically maintain these hashmaps for us.
4. If  $k$  is greater than 2, then we will make recursive calls to  $f$  and  $p$  according to the transition function:
  - $f(k) = f(k-1) + f(k-2) + 2 * p(k-1)$
  - $p(k) = p(k-1) + f(k-2)$
5.  $f(n)$  will be returned once all recursive calls are finished.

### Implementation

C++:

```
class Solution {
public:
    int MOD = 1'000'000'007;
    unordered_map<int, long> f_cache;
    unordered_map<int, long> p_cache;

    long p(int n) {
        if (p_cache.find(n) != p_cache.end()) {
```

```

        return p_cache[n];
    }
    long val;
    if (n == 2) {
        val = 1L;
    } else {
        val = (p(n - 1) + f(n - 2)) % MOD;
    }
    p_cache[n] = val;
    return val;
};

```

```

long f(int n) {
    if (f_cache.find(n) != f_cache.end()) {
        return f_cache[n];
    }
    long val;
    if (n == 1) {
        val = 1L;
    } else if (n == 2) {
        val = 2L;
    } else {
        val = (f(n - 1) + f(n - 2) + 2 * p(n - 1)) % MOD;
    }
    f_cache[n] = val;
    return val;
};

```

```

int numTilings(int n) {
    return static_cast<int>(f(n));
}

```

};

### Complexity Analysis

Let  $N$  be the width of the board.

- Time complexity:  $O(N)$

From top ( $N$ ) to bottom (1), there will be  $N$  non-memoized recursive calls to  $f$  and to  $p$ , where each non-memoized call requires constant time. Thus,  $O(2 \cdot N)$  time is required for the non-memoized calls.

Furthermore, there will be  $2 \cdot N$  memoized calls to  $f$  and  $N$  memoized calls to  $p$ , where each memoized call also requires constant time. Thus  $O(3 \cdot N)$  time is required for the memoized calls.

This leads to a time complexity of  $O(2 \cdot N + 3 \cdot N) = O(N)$ .

- Space complexity:  $O(N)$

Each recursion call stack will contain at most  $N$  layers. Also, each hashmap will use  $O(N)$  space. Together this results in  $O(N)$  space complexity.

---

## Approach 2: Dynamic Programming (Bottom-up)

### Intuition

One of the drawbacks to the previous top-down DP solution is that it uses a recursive call stack which requires additional time and space to maintain. When we know that all of the subproblems (i.e.  $f(1)$  through  $f(n - 1)$ ) must be solved and there is a logical order to the subproblems (i.e.  $f(1)$  must be solved before  $f(2)$ , and  $f(2)$  before ...  $f(n - 1)$ ), then bottom-up DP will generally be more efficient than top-down DP because it will solve the same number of subproblems and do so without maintaining a call stack. Different from the previous *top-down* solution, this solution will be in a *bottom-up* fashion. As such, we will start by calculating the base case (when  $n$  is very small, like 0, 1, 2 etc.), then move to the next case (when  $n$  grows to a larger number) and gradually get the result for the final case  $f(n)$ .

### Algorithm

1. Create two arrays,  $f$  and  $p$ , of size  $n + 1$ , where  $f(k)$  represents the number of ways to **fully cover a board** of width  $k$  and  $p(k)$  represents the number of ways to **partially cover a board** of width  $k$  (as described in the overview).
2. Initialize  $f$  and  $p$  according to the following base cases:
  - $f(1) = 1$  because to **fully cover a board** of width 1, there is only one way, add one vertical domino.
  - $f(2) = 2$  because to **fully cover a board** of width 2, there are two ways, either add two horizontal dominos or add two vertical dominos.
  - $p(2) = 1$  because to **partially cover a board** of width 2, there is only one way using an L-shaped tromino (leave the upper-right corner uncovered).
3. Iterate  $k$  from 2 to  $n$  (inclusive) and at each iteration update  $f$  and  $p$  according to the transition functions we derived in the overview:
  - $f(k) = f(k - 1) + f(k - 2) + 2 * p(k - 1)$
  - $p(k) = p(k - 1) + f(k - 2)$
4. Return  $f(n)$  which now represents the number of ways to **fully cover a board** of width  $n$ .

### Implementation

C++:

```
class Solution {
public:
    int numTilings(int n) {
        int MOD = 1'000'000'007;
        // handle base case scenarios
        if (n <= 2) {
            return n;
        }
        // f[k]: number of ways to "fully cover a board" of width k
        long f[n + 1];
        // p[k]: number of ways to "partially cover a board" of width k
        long p[n + 1];
        // initialize f and p with results for the base case scenarios
        f[1] = 1L;
        f[2] = 2L;
        p[2] = 1L;
        for (int k = 3; k < n + 1; ++k) {
            f[k] = (f[k - 1] + f[k - 2] + 2 * p[k - 1]) % MOD;
            p[k] = (p[k - 1] + f[k - 2]) % MOD;
        }
        return static_cast<int>(f[n]);
    }
};
```



### Complexity Analysis

Let  $N$  be the width of the board.

- Time complexity:  $O(N)$

Array iteration requires  $N - 2$  iterations where each iteration takes constant time.

- Space complexity:  $O(N)$

Two arrays of size  $N + 1$  are used to store the number of ways to fully and partially tile boards of various widths between 1 and  $N$ .

---

### Approach 3: Dynamic Programming (Bottom-up, space optimization)

#### Intuition

In the previous approach, we used arrays of length  $N + 1$  to store the number of ways to fully and partially cover boards of varying width. In total  $O(N)$  space was used for this purpose. However, based on the *transition function*, we find that only  $f(n - 1)$ ,  $f(n - 2)$  and  $p(n - 1)$  are used in each iteration. One of the benefits of bottom-up dynamic programming is that we have some control over how long to cache the result of each subproblem. For instance if the result of a subproblem like  $f(3)$  is only needed to find the result for  $f(4)$  and  $f(5)$ , then we no longer need to store the result for  $f(3)$  after  $f(4)$  and  $f(5)$  have been calculated.

Thus, instead of using an entire array to store this information, we can use just 3 numeric variables to store the result of  $f(k - 1)$ ,  $f(k - 2)$ , and  $p(k - 1)$ . In this way, we can further optimize the space complexity to  $O(1)$ !

## Algorithm

1. Create three numerical variables,

- `fCurrent` represents  $f(k-1)$ , where  $f(k)$  is number of ways to **fully cover a board** of width  $k$ .
- `pCurrent` represents  $p(k-1)$ , where  $p(k)$  is number of ways to **partially cover a board** (as described in the **Overview** section) of width  $k$ .
- `fPrevious` represents  $f(k-2)$ .

Since `k` starts from 3, the three variables will have the following initial values:

- `fCurrent = 2` (i.e.  $f(k-1) = f(2) = 2$ ), because there are two ways to **fully tile a board** of width 2: add two vertical dominos or add two horizontal dominos.
- `pCurrent = 1` (i.e.  $p(k-1) = p(2) = 1$ ), because there is exactly one way to **partially cover a board** with one column.
  - Besides,  $p(1) = 0$ , because it is impossible to **partially cover a board** with one column.
- `fPrevious = 1` (i.e.  $f(k-2) = f(1) = 1$ ), because there is exactly one way to **fully cover a board** of width 1: add one vertical domino.

2. Iterate  $k$  from 3 to  $n$  (inclusive) and at each iteration update the above three variables according to the transition functions:

- $f(k) = f(k-1) + f(k-2) + 2 * p(k-1)$
- $p(k) = p(k-1) + f(k-2)$

By applying the optimization mentioned in the *intuition* section, we get the following:

- `fCurrent = fCurrent + fPrevious + 2 * pCurrent`
- `pCurrent = pCurrent + fPrevious`
- `fPrevious = fCurrent` (use the value of `fCurrent` before its update in the first step)

3. Return `fCurrent` which now represents the number of ways to **fully cover a board** of width  $n$ .

## Implementation

C++:

```
class Solution {
public:
    int numTilings(int n) {
        int MOD = 1'000'000'007;
        if (n <= 2) {
            return n;
        }
        long fPrevious = 1L;
        long fCurrent = 2L;
        long pCurrent = 1L;
        for (int k = 3; k < n + 1; ++k) {
```

```

    long tmp = fCurrent;
    fCurrent = (fCurrent + fPrevious + 2 * pCurrent) % MOD;
    pCurrent = (pCurrent + fPrevious) % MOD;
    fPrevious = tmp;
}
return static_cast<int>(fCurrent);
}
};

```

### Complexity Analysis

- Time complexity:  $O(N)$

Array iteration takes  $O(N)$  time where  $N$  is the width of the board.

- Space complexity:  $O(1)$

Only a constant number of numeric ( `long` / `int` ) variables were used.

## Approach 4: Matrix Exponentiation

### Intuition

Matrix Exponentiation? It's a big word, but don't let it scare you! It's a simple math technique based on the previous dynamic programming idea. The only prerequisite knowledge you need is [Matrix Multiplication](#). Trust me, this approach will help turn matrix exponentiation from a big word into a piece of cake.

Previously, we used the following *transition function*:

$$\begin{aligned}f(k) &= f(k-1) + f(k-2) + 2 * p(k-1) \\p(k) &= p(k-1) + f(k-2)\end{aligned}$$

Now, let's take this relationship and convert it to a matrix representation as shown below. We will discuss why it's represented this way later, for now, let's just follow along.

$$\begin{bmatrix} f(k) \\ f(k-1) \\ p(k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(k-1) \\ f(k-2) \\ p(k-1) \end{bmatrix}$$

Notice, the 1, 1, 2 in the first row corresponds to the one  $f(k-1)$ , one  $f(k-2)$ , and two  $p(k-1)$  that add up to  $f(k)$ .

By replacing  $k$  with  $k-1$ , we will get:

$$\begin{bmatrix} f(k-1) \\ f(k-2) \\ p(k-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(k-2) \\ f(k-3) \\ p(k-2) \end{bmatrix}$$

Notice, the leftmost matrix is now the same as the rightmost matrix from the first step.

Take the equation above and insert it in the first equation. This gives us the following equation:

$$\begin{bmatrix} f(k) \\ f(k-1) \\ p(k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(k-2) \\ f(k-3) \\ p(k-2) \end{bmatrix}$$

This process can be repeated until the base case is reached:

This process can be repeated until the base case is reached:

$$\begin{bmatrix} f(k) \\ f(k-1) \\ p(k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \dots k-2 \text{ times} \dots * \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(2) \\ f(1) \\ p(2) \end{bmatrix}$$

We can use the equation above to find the relationship between the target case ( $f(k)$ ) and the base cases. Take the  $3 \times 3$  matrix and multiply it by itself  $k-2$  times, and we will arrive at the solution for  $f(k)$ .

Now we've covered the basic idea of matrix exponentiation; but why we are doing it like this and how did this decomposition of the *transition functions* come up in the first place?

First, matrix exponentiation is a technique, just like binary search, depth-first search, etc. It's not supposed to be something you come up with on your own, instead, it is something that you can learn and apply elsewhere.

Now, let's follow up with some experimental conclusions about matrix exponentiation:

*First Matrix = Second Matrix \* Third Matrix*

$$\begin{bmatrix} f(k) \\ f(k-1) \\ p(k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(k-1) \\ f(k-2) \\ p(k-1) \end{bmatrix}$$

- It's often possible to solve a **Dynamic Programming (DP)** question with **Matrix Exponentiation** when there is a DP *transition function*
- The integer matrix (the second matrix) has to be a square matrix ( $m \times n$ , where  $m = n$ , we will use  $m \times m$  for simplicity). This is because only the square matrix doesn't change its dimension after multiplication, meaning it will still be  $m \times m$  after multiplication.
- The dimension of the third matrix has to be  $m \times 1$ , it should include all used elements on the right side of the *transition function*. In our case, they are  $f(k-1)$ ,  $f(k-2)$  and  $p(k-1)$ .
- The first matrix will be the same as the third matrix, except that we will use  $k+1$  to replace  $k$ .
- With knowledge of the first matrix, the third matrix, *transition function*, and the fact that the second matrix must be a square matrix, you can fill in the values of the second matrix on your own using basic matrix multiplication knowledge.
  - $f(k) = 1 * f(k-1) + 1 * f(k-2) + 2 * p(k-1)$  (transition function)
  - $f(k-1) = 1 * f(k-1) + 0 * f(k-2) + 0 * p(k-1)$
  - $p(k) = 0 * f(k-1) + 1 * f(k-2) + 1 * p(k-1)$  (transition function)

With the above knowledge, a good practice question will be [509. Fibonacci Number](#). See if this time you can come up with a matrix exponentiation solution by yourself.

## Algorithm

1. Again, we use the same definition of  $f$  and  $p$  as previously described in the **Overview** section
  - $f(k)$ : The number of ways to **fully cover a board** of width  $k$
  - $p(k)$ : The number of ways to **partially cover a board** of width  $k$
2. Pre-calculate base cases:
  - $f(1) = 1$  because to **fully cover a board** of width 1, there is only one way, add one vertical domino.
  - $f(2) = 2$  because to **fully cover a board** of width 2, there are two ways, either add two horizontal dominos or add two vertical dominos.
  - $p(2) = 1$  because to **partially cover a board** of width 2, there is only one way using a L-shaped tromino (leave the upper-right corner uncovered).
3. Prepare the square matrix based on the transition function.
4. Create a function/method to calculate **matrix multiplication**.
5. Calculate the square matrix multiplication  $k-2$  times iteratively.
6. Take the resulting square matrix from the above step and calculate  $f(k)$  as shown below:

$$\begin{bmatrix} f(k) \\ f(k-1) \\ p(k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \dots k-2 \text{ times } \dots * \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(2) \\ f(1) \\ p(2) \end{bmatrix}$$

## Implementation

C++:

```
class Solution {
public:
    // To avoid confusing point syntax, we use 2d-vector instead
    vector<vector<long>> SQ_MATRIX { // Initialize square matrix
        {1, 1, 2},
        {1, 0, 0},
        {0, 1, 1},
    };
    int MOD = 1'000'000'007;
    int SIZE = 3; // Width/Lenght of the square matrix

    // Return product of 2 square matrices.
    vector<vector<long>> matrixProduct(vector<vector<long>> m1,
        vector<vector<long>> m2) {

        // Result matrix `ans` will also be a square matrix with same dimension
        vector<vector<long>> ans = SQ_MATRIX;
        for (int row = 0; row < SIZE; ++row) {
            for (int col = 0; col < SIZE; ++col) {
                long curSum = 0;
                for (int k = 0; k < SIZE; ++k) {
                    curSum = (curSum + m1[row][k] * m2[k][col]) % MOD;
                }
            }
        }
        return ans;
    }
};
```

```

        }
        ans[row][col] = curSum;
    }
}
return ans;
}

// Return answer after n times matrix multiplication.
int matrixExpo(int n) {
    vector<vector<long>> cur = SQ_MATRIX;
    for (int i = 1; i < n; ++i) {
        cur = matrixProduct(cur, SQ_MATRIX);
    }
    // The answer will be  $cur[0][0] * f(2) + cur[0][1] * f(1) + cur[0][2] * p(2)$ 
    return static_cast<int>((cur[0][0] * 2 + cur[0][1] * 1 + cur[0][2] * 1) % MOD);
}

int numTilings(int n) {
    // Handle base cases
    if (n <= 2) {
        return n;
    }
    return matrixExpo(n-2);
}
};

```

### Complexity Analysis

Let  $N$  be the width of the board.

- Time complexity:  $O(N)$

We need to perform matrix multiplication of  $3 \times 3$  matrix  $N - 2$  times which will take  $O(N)$  time. This dominates the time costs of the rest of operations.

- Space complexity:  $O(1)$

We only used a  $3 \times 3$  matrix and a few other numeric variables.

---



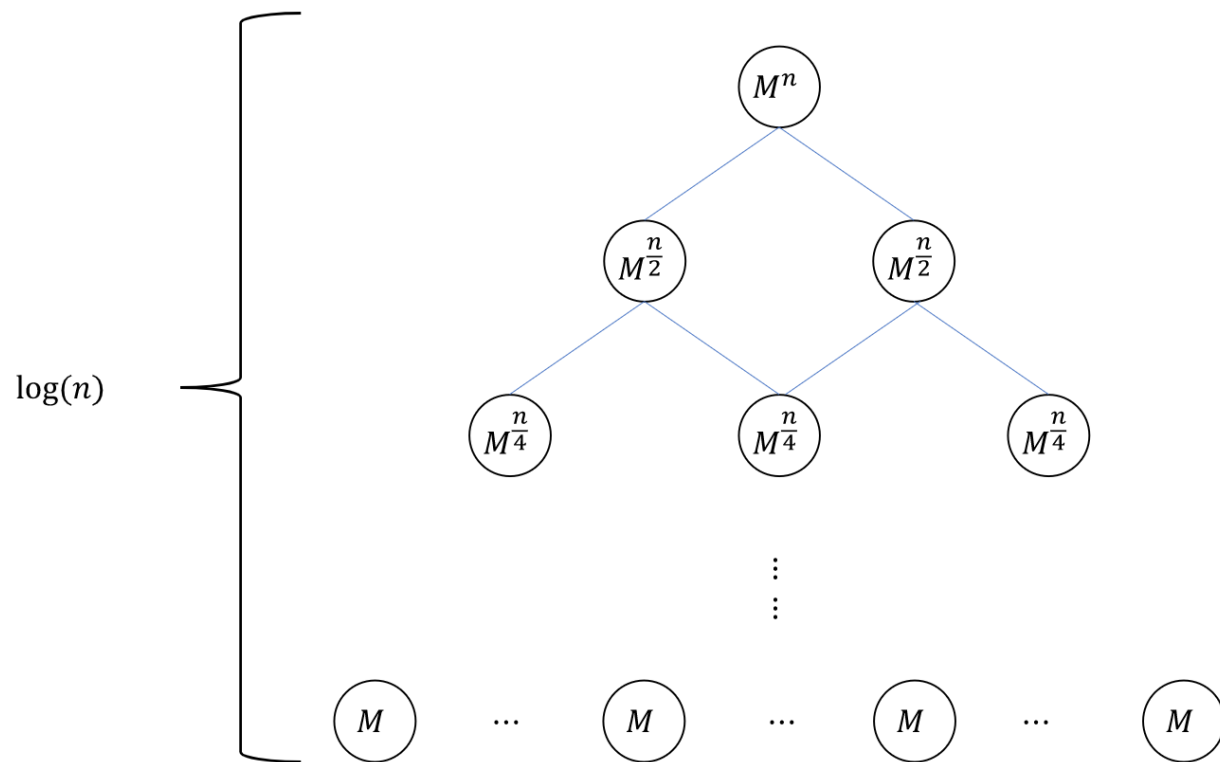
## Approach 5: Matrix Exponentiation (time optimization, space/time trade off)

### Intuition

In the previous method, we calculate the matrix  $M$  to the power of  $n$  in an iterative fashion. This iterative approach is equivalent to calculating  $M$  times  $M$  times  $M$  ... for  $n$  times. This is why the previous approach requires linear time.

However, when finding the result of some number (or matrix in this case) to the power of another number, there is an optimization we can use to perform the operation in logarithmic time. If you are unfamiliar with this optimization, we encourage you to try [50. Pow\(x, n\)](#). Which uses the same optimization but for a less complicated problem.

To use this optimization, we must first observe that  $M^n$  is equivalent to  $M^{n/2} \cdot M^{n/2}$  which is equivalent to  $(M^{n/4} \cdot M^{n/4}) \cdot (M^{n/4} \cdot M^{n/4})$ . Thus, we can obtain  $M^n$  by recursively dividing the power by 2 and multiplying the results until we reach  $M^1$ , which is just  $M$ . As long as  $n$  is an even number, this process can be represented as the following binary tree.



What if  $n$  is an odd number, you may ask. Well, if  $n$  is odd then  $n - 1$  will be even, we can apply the above process on  $n - 1$  and take the result of  $pow(M, n - 1)$  and multiply it with the base. This will result in  $pow(M, n) = pow(M, n - 1) * M$ .

One basic fact about the binary tree is that the binary tree is a recursive data structure. By using recursion, we can reduce the time complexity down to the height of the tree, which is  $\log(n)$ . At the same time, we will use a cache to avoid repeat calculations. As you may have noticed, the same number is being used multiple times during the calculation.

### Algorithm

The algorithm used here is very similar to the previous approach. The only difference is, in step 4, we will use **recursion** to perform the square matrix multiplication rather than iteration.

- Again, we use the same definition of **f** and **p** as previously described in the **Overview** section
  - $f(k)$ : The number of ways to **fully cover a board** of width  $k$
  - $p(k)$ : The number of ways to **partially cover a board** of width  $k$
- Pre-calculate base cases:
  - $f(1) = 1$  because to **fully cover a board** of width 1, there is only one way, add one vertical domino.
  - $f(2) = 2$  because to **fully cover a board** of width 2, there are two ways, either add two horizontal dominos or add two vertical dominos.
  - $p(2) = 1$  because to **partially cover a board** of width 2, there is only one way using a L-shaped tromino (leave the upper-right corner uncovered).
- Prepare the square matrix based on the previous description.
- Create a function/method to calculate matrix multiplication **recursively**.
- Calculate the square matrix multiplication  $k-2$  times recursively. Meanwhile, store the intermediate results in a map to avoid repeat calculations (caching).
- Take the square matrix that results from the above step and calculate  $f(k)$  as shown below:

$$\begin{bmatrix} f(k) \\ f(k-1) \\ p(k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \dots k-2 \text{ times} \dots * \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} * \begin{bmatrix} f(2) \\ f(1) \\ p(2) \end{bmatrix}$$

### Implementation

C++:

```
class Solution {
public:
    int MOD = 1'000'000'007;
    vector<vector<long>>> SQ_MATRIX { // Initialize square matrix
        {1, 1, 2},
        {1, 0, 0},
        {0, 1, 1},
    };
    int SIZE = 3; // Width/Length of square matrix
    unordered_map<int, vector<vector<long>>>> cache;

    // Return product of 2 square matrices
    vector<vector<long>>> matrixProduct(
```

```

vector<vector<long>> m1, vector<vector<long>> m2) {

    // Result matrix `ans` will also be a square matrix with same dimension
    vector<vector<long>> ans = SQ_MATRIX;
    for (int row = 0; row < SIZE; ++row) {
        for (int col = 0; col < SIZE; ++col) {
            long curSum = 0;
            for (int k = 0; k < SIZE; ++k) {
                curSum = (curSum + m1[row][k] * m2[k][col]) % MOD;
            }
            ans[row][col] = curSum;
        }
    }
    return ans;
}

// Return pow(SQ_MATRIX, n)
vector<vector<long>> matrixExpo(int n) {
    if (cache.find(n) != cache.end()) {
        return cache[n];
    }
    vector<vector<long>> cur = SQ_MATRIX;
    if (n == 1) { // base case
        cur = SQ_MATRIX;
    } else if (n % 2) { // If `n` is odd
        cur = matrixProduct(matrixExpo(n - 1), SQ_MATRIX);
    } else { // If `n` is even
        cur = matrixProduct(matrixExpo(n / 2), matrixExpo(n / 2));
    }
    cache[n] = cur;
    return cur;
}

int numTilings(int n) {
    if (n <= 2) { // Handle base cases
        return n;
    }
    // The answer will be cur[0][0] * f(2) + cur[0][1] * f(1) + cur[0][2] * p(2)
    vector<long> ans = matrixExpo(n - 2)[0];
    return (ans[0] * 2 + ans[1] * 1 + ans[2] * 1) % MOD;
}
};

```

## Complexity Analysis

Let  $N$  be the width of the board.

- Time complexity:  $O(\log N)$

With the use of recursion and memoization, we only need to make one calculation per level of the recursion tree. As previously shown, the number of matrix multiplications can be further reduced down to  $O(\log N)$ .

- Space complexity:  $O(\log N)$

Stack space of  $O(\log N)$  will be used due to recursion. Also, an extra  $O(\log N)$  space will be used for caching/memoization during recursion, since we used a map to store the intermediate results, where the key is an integer and the value is a 3 by 3 matrix. Together they will take  $O(\log N)$  space.

## Approach 6: Math optimization (Fibonacci sequence like)

Hint: Can you take the transition functions from *Approach 2* and get rid of  $p$ ?

### Intuition

This is the ultimate optimization for this question, there won't be any advance to the time or space complexity, but it will make the code cleaner and the logic much more clear. It's all based on simple math operations and 5 other solutions we discussed previously. Let's take a closer look at the derivation below.

From previous Dynamic Programming transition function, we get  
 $f(k) = f(k-1) + f(k-2) + 2 * p(k-1)$ , let's call it equation 1  
 $p(k) = p(k-1) + f(k-2)$ , (equation 2)

Take equation 1 and replace  $k$  by  $k-1$ , then will get (equation 3):  
 $f(k-1) = f(k-2) + f(k-3) + 2 * p(k-2)$

Take equation 2 and move  $p(k-1)$  to the left side of equal sign, we get (equation 4):  
 $p(k) - p(k-1) = f(k-2)$

Take equation 4 and replace  $k$  by  $k-1$ , then will get (equation 5):  
 $p(k-1) - p(k-2) = f(k-3)$

Subtract equation 3 from equation 1 to get  
 $f(k) - f(k-1) = f(k-1) + f(k-2) + 2 * p(k-1) - f(k-2) - f(k-3) - 2 * p(k-2)$

Simplify above we will get  
 $f(k) - f(k-1) = f(k-1) - f(k-3) + 2 * (p(k-1) - p(k-2))$

Now, use equation 5 to substitute  $f(k-3)$  into the above equation  
 $f(k) - f(k-1) = f(k-1) - f(k-3) + 2 * f(k-3) = f(k-1) + f(k-3)$

Move  $f(k-1)$  to the right side of the equation, we will get  
 $f(k) = 2 * f(k-1) + f(k-3)$

Does this final result remind you of something? Yes, the Fibonacci Sequence had a similar *transition function*. Now, we can remove  $p(k)$  from our *transition function*, and focus solely on  $f(k)$ . With this new *transition function*, we can re-apply every single method we mentioned earlier. Here we will observe a replica of *approach 2*. For practice, you can try out the method mentioned in *Approach 5* with this new *transition function*.

## Algorithm

This algorithm is very similar to *Approach 3: Space Optimized Bottom-up DP*, but this time, we only need one transition function, and we no longer need to use  $p$ .

1. Create three numeric variables.

- `fCurrent` represents  $f(k - 1)$ , where  $f(k)$  is number of ways to **fully cover a board** of width  $k$ .
- `fPrevious` represents  $f(k - 2)$ .
- `fBeforePrevious` represents  $f(k - 3)$ .

Since  $k$  starts from 4, the three variables will have the following initial values:

- $fCurrent = 5$  (i.e.  $f(k - 1) = f(3) = 5$ ), because there are five ways to **fully tile a board** of width 3 (as shown in the first animation in the **Overview** section).
- $fPrevious = 2$  (i.e.  $f(k - 2) = f(2) = 2$ ), because there are two ways to **fully tile a board** of width 2 (as shown in the first animation in the **Overview** section).
- $fBeforePrevious = 1$  (i.e.  $f(k - 3) = f(1) = 1$ ), because there is exactly one way to **fully cover a board** of width 1: add one vertical domino

2. Iterate  $k$  from 4 to  $n$  and at each iteration update the above three variables according to the transition function mentioned previously:  $f(k) = 2 * f(k - 1) + f(k - 3)$

3. Return `fCurrent` which now represents the number of ways to fully tile a board of width  $n$ .

## Implementation

C++:

```
class Solution {
public:
    int numTilings(int n) {
        int MOD = 1'000'000'007;
        if (n <= 2) {
            return n;
        }
        long fCurrent = 5L;
        long fPrevious = 2L;
        long fBeforePrevious = 1L;
        for (int k = 4; k < n + 1; ++k) {
            long tmp = fPrevious;
            fPrevious = fCurrent;
            fCurrent = (2 * fCurrent + fBeforePrevious) % MOD;
            fBeforePrevious = tmp;
        }
        return static_cast<int>(fCurrent);
    }
};
```

Don't forget to try out the  $O(\log N)$  matrix exponentiation solution with this new *transition function*.

### Complexity Analysis

- Time complexity:  $O(N)$

Array iteration takes  $O(N)$  time, where  $N$  is the width of the board.

- Space complexity:  $O(1)$

Only a constant number of numeric ( `long` / `int` ) variables were used.