Problem 2189: Number of Ways to Build House of Cards

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given an integer

n

representing the number of playing cards you have. A

house of cards

meets the following conditions:

A

house of cards

consists of one or more rows of

triangles

and horizontal cards.

Triangles

are created by leaning two cards against each other.

One card must be placed horizontally between all adjacent

triangles in a row.

Any triangle on a row higher than the first must be placed on a horizontal card from the previous row.

Each triangle is placed in the

leftmost

available spot in the row.

Return

the number of

distinct

house of cards

you can build using

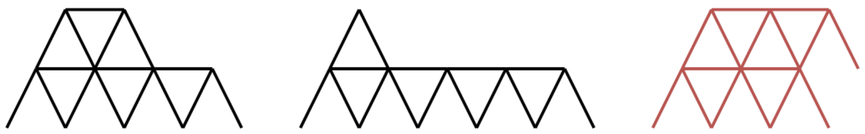
all

n

cards.

Two houses of cards are considered distinct if there exists a row where the two houses contain a different number of cards.

Example 1:



Input:

n = 16

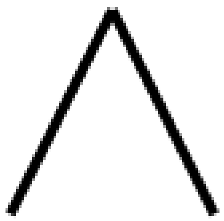
Output:

2

Explanation:

The two valid houses of cards are shown. The third house of cards in the diagram is not valid because the rightmost triangle on the top row is not placed on top of a horizontal card.

Example 2:



Input:

n = 2

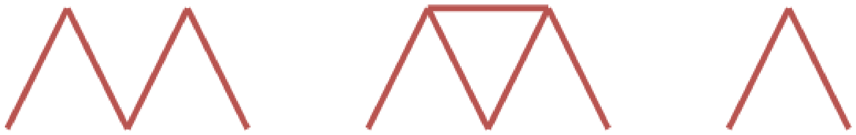
Output:

1

Explanation:

The one valid house of cards is shown.

Example 3:



Input:

n = 4

Output:

0

Explanation:

The three houses of cards in the diagram are not valid. The first house of cards needs a horizontal card placed between the two triangles. The second house of cards uses 5 cards. The third house of cards uses 2 cards.

Constraints:

1 <= n <= 500

Code Snippets

C++:

```
class Solution {
public:
int houseOfCards(int n) {

}
};
```

Java:

```
class Solution {
public int houseOfCards(int n) {
}
}
```

Python3:

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

Python:

```
class Solution(object):
def houseOfCards(self, n):
"""
:type n: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} n
* @return {number}
*/
var houseOfCards = function(n) {

};
```

TypeScript:

```
function houseOfCards(n: number): number {

};
```

C#:

```
public class Solution {
public int HouseOfCards(int n) {

}
}
```

C:

```
int houseOfCards(int n) {

}
```

Go:

```
func houseOfCards(n int) int {

}
```

Kotlin:

```
class Solution {
fun houseOfCards(n: Int): Int {

}
}
```

Swift:

```
class Solution {
func houseOfCards(_ n: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn house_of_cards(n: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} n
# @return {Integer}
def house_of_cards(n)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @return Integer
*/
function houseOfCards($n) {

}
}
```

Dart:

```
class Solution {
int houseOfCards(int n) {

}
}
```

Scala:

```
object Solution {
def houseOfCards(n: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec house_of_cards(n :: integer) :: integer
def house_of_cards(n) do

end
end
```

Erlang:

```
-spec house_of_cards(N :: integer()) -> integer().
house_of_cards(N) ->
.
```

Racket:

```
(define/contract (house-of-cards n)
(-> exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Number of Ways to Build House of Cards
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int houseOfCards(int n) {

}
};
```

Java Solution:

```
/**
* Problem: Number of Ways to Build House of Cards
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int houseOfCards(int n) {
}
}
```

Python3 Solution:

```
"""
Problem: Number of Ways to Build House of Cards
Difficulty: Medium
Tags: dp, math

Approach: Dynamic programming with memoization or tabulation
Time Complexity: O(n * m) where n and m are problem dimensions
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def houseOfCards(self, n: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def houseOfCards(self, n):
"""
:type n: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Number of Ways to Build House of Cards
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
/**
* @param {number} n
* @return {number}
*/
var houseOfCards = function(n) {

};
```

TypeScript Solution:

```
/**
* Problem: Number of Ways to Build House of Cards
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

function houseOfCards(n: number): number {

};
```

C# Solution:

```
/*
* Problem: Number of Ways to Build House of Cards
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int HouseOfCards(int n) {

}
}
```

C Solution:

```
/*
* Problem: Number of Ways to Build House of Cards
* Difficulty: Medium
* Tags: dp, math
*
* Approach: Dynamic programming with memoization or tabulation
* Time Complexity: O(n * m) where n and m are problem dimensions
* Space Complexity: O(n) or O(n * m) for DP table
*/

int houseOfCards(int n) {

}
```

Go Solution:

```
// Problem: Number of Ways to Build House of Cards
// Difficulty: Medium
// Tags: dp, math
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
// Space Complexity: O(n) or O(n * m) for DP table

func houseOfCards(n int) int {

}
```

Kotlin Solution:

```
class Solution {
fun houseOfCards(n: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func houseOfCards(_ n: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Number of Ways to Build House of Cards
// Difficulty: Medium
// Tags: dp, math
//
// Approach: Dynamic programming with memoization or tabulation
// Time Complexity: O(n * m) where n and m are problem dimensions
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn house_of_cards(n: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @return {Integer}
def house_of_cards(n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @return Integer
*/
function houseOfCards($n) {

}
}
```

Dart Solution:

```
class Solution {
int houseOfCards(int n) {

}
}
```

Scala Solution:

```
object Solution {
def houseOfCards(n: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec house_of_cards(n :: integer) :: integer
def house_of_cards(n) do

end
end
```

Erlang Solution:

```
-spec house_of_cards(N :: integer()) -> integer().
house_of_cards(N) ->
.
```

Racket Solution:

```
(define/contract (house-of-cards n)
(-> exact-integer? exact-integer?)
)
```