Problem 1866: Number of Ways to Rearrange Sticks With K Sticks Visible

Problem Information

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

Problem Description

There are

n

uniquely-sized sticks whose lengths are integers from

1

to

n

. You want to arrange the sticks such that

exactly

k

sticks are

visible

from the left. A stick is

visible

from the left if there are no

longer

sticks to the

left

of it.

For example, if the sticks are arranged

[

1

,

3

,2,

5

,4]

, then the sticks with lengths

1

,

3

, and

5

are visible from the left.

Given

n

and

k

, return

the

number

of such arrangements

. Since the answer may be large, return it

modulo

$10^9 + 7$

.

Example 1:

Input:

n = 3, k = 2

Output:

3

Explanation:

[

1

,

3

,2], [

2

,

3

,1], and [

2

,1,

3

] are the only arrangements such that exactly 2 sticks are visible. The visible sticks are underlined.

Example 2:

Input:

n = 5, k = 5

Output:

1

Explanation:

[

1

,

2

,

3

,

4

,

5

] is the only arrangement such that all 5 sticks are visible. The visible sticks are underlined.

Example 3:

Input:

n = 20, k = 11

Output:

647427950

Explanation:

There are 647427950 (mod 10

9

+ 7) ways to rearrange the sticks such that exactly 11 sticks are visible.

Constraints:

1 <= n <= 1000

1 <= k <= n

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int rearrangeSticks(int n, int k) {

}
}
```

Python3:

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

Python:

```
class Solution(object):
def rearrangeSticks(self, n, k):
"""
:type n: int
```

```
:type k: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number} k
* @return {number}
*/
var rearrangeSticks = function(n, k) {

};
```

TypeScript:

```
function rearrangeSticks(n: number, k: number): number {

};
```

C#:

```
public class Solution {
public int RearrangeSticks(int n, int k) {

}
}
```

C:

```
int rearrangeSticks(int n, int k) {

}
```

Go:

```
func rearrangeSticks(n int, k int) int {

}
```

Kotlin:

```
class Solution {
fun rearrangeSticks(n: Int, k: Int): Int {

}
}
```

Swift:

```
class Solution {
func rearrangeSticks(_ n: Int, _ k: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn rearrange_sticks(n: i32, k: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer} k
# @return {Integer}
def rearrange_sticks(n, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer $k
* @return Integer
*/
function rearrangeSticks($n, $k) {

}
```

```
}
```

Dart:

```
class Solution {
int rearrangeSticks(int n, int k) {

}
}
```

Scala:

```
object Solution {
def rearrangeSticks(n: Int, k: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec rearrange_sticks(n :: integer, k :: integer) :: integer
def rearrange_sticks(n, k) do

end
end
```

Erlang:

```
-spec rearrange_sticks(N :: integer(), K :: integer()) -> integer().
rearrange_sticks(N, K) ->
.
```

Racket:

```
(define/contract (rearrange-sticks n k)
(-> exact-integer? exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
* Difficulty: Hard
* 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 rearrangeSticks(int n, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
* Difficulty: Hard
* 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 rearrangeSticks(int n, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
Difficulty: Hard
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 rearrangeSticks(self, n: int, k: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
* Difficulty: Hard
* 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
* @param {number} k
* @return {number}
*/
var rearrangeSticks = function(n, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
* Difficulty: Hard
* 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 rearrangeSticks(n: number, k: number): number {

};
```

C# Solution:

```
/*
* Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
* Difficulty: Hard
* 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 RearrangeSticks(int n, int k) {

}
}
```

C Solution:

```
/*
* Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
* Difficulty: Hard
* 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 rearrangeSticks(int n, int k) {

}
```

Go Solution:

```
// Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
// Difficulty: Hard
// 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 rearrangeSticks(n int, k int) int {

}
```

Kotlin Solution:

```
class Solution {
fun rearrangeSticks(n: Int, k: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func rearrangeSticks(_ n: Int, _ k: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Number of Ways to Rearrange Sticks With K Sticks Visible
// Difficulty: Hard
// 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 rearrange_sticks(n: i32, k: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer} k
# @return {Integer}
def rearrange_sticks(n, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer $k
* @return Integer
*/
function rearrangeSticks($n, $k) {

}
}
```

Dart Solution:

```
class Solution {
int rearrangeSticks(int n, int k) {
```

```
}
}
```

Scala Solution:

```
object Solution {
def rearrangeSticks(n: Int, k: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec rearrange_sticks(n :: integer, k :: integer) :: integer
def rearrange_sticks(n, k) do

end
end
```

Erlang Solution:

```
-spec rearrange_sticks(N :: integer(), K :: integer()) -> integer().
rearrange_sticks(N, K) ->
.
```

Racket Solution:

```
(define/contract (rearrange-sticks n k)
(-> exact-integer? exact-integer? exact-integer?)
)
```