Problem 377: Combination Sum IV

Problem Information

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

Problem Description

Given an array of

distinct

integers

nums

and a target integer

target

, return

the number of possible combinations that add up to

target

.

The test cases are generated so that the answer can fit in a

32-bit

integer.

Example 1:

Input:

nums = [1,2,3], target = 4

Output:

7

Explanation:

The possible combination ways are: (1, 1, 1, 1) (1, 1, 2) (1, 2, 1) (1, 3) (2, 1, 1) (2, 2) (3, 1)
Note that different sequences are counted as different combinations.

Example 2:

Input:

nums = [9], target = 3

Output:

0

Constraints:

1 <= nums.length <= 200

1 <= nums[i] <= 1000

All the elements of

nums

are

unique

.

1 <= target <= 1000

Follow up:

What if negative numbers are allowed in the given array? How does it change the problem? What limitation we need to add to the question to allow negative numbers?

Code Snippets

C++:

```
class Solution {
public:
int combinationSum4(vector<int>& nums, int target) {

}
};
```

Java:

```
class Solution {
public int combinationSum4(int[] nums, int target) {

}
}
```

Python3:

```
class Solution:
def combinationSum4(self, nums: List[int], target: int) -> int:
```

Python:

```
class Solution(object):
def combinationSum4(self, nums, target):
"""
:type nums: List[int]
:type target: int
:rtype: int
```

```
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @param {number} target
* @return {number}
*/
var combinationSum4 = function(nums, target) {

};
```

TypeScript:

```
function combinationSum4(nums: number[], target: number): number {

};
```

C#:

```
public class Solution {
public int CombinationSum4(int[] nums, int target) {

}
}
```

C:

```
int combinationSum4(int* nums, int numsSize, int target) {

}
```

Go:

```
func combinationSum4(nums []int, target int) int {

}
```

Kotlin:

```
class Solution {
fun combinationSum4(nums: IntArray, target: Int): Int {

}
}
```

Swift:

```
class Solution {
func combinationSum4(_ nums: [Int], _ target: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn combination_sum4(nums: Vec<i32>, target: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer} target
# @return {Integer}
def combination_sum4(nums, target)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $target
* @return Integer
*/
function combinationSum4($nums, $target) {

}
}
```

Dart:

```
class Solution {
int combinationSum4(List<int> nums, int target) {

}
}
```

Scala:

```
object Solution {
def combinationSum4(nums: Array[Int], target: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec combination_sum4(nums :: [integer], target :: integer) :: integer
def combination_sum4(nums, target) do

end
end
```

Erlang:

```
-spec combination_sum4(Nums :: [integer()], Target :: integer()) ->
integer().
combination_sum4(Nums, Target) ->
.
```

Racket:

```
(define/contract (combination-sum4 nums target)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Combination Sum IV
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int combinationSum4(vector<int>& nums, int target) {

}
};
```

Java Solution:

```
/**
* Problem: Combination Sum IV
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public int combinationSum4(int[] nums, int target) {

}
}
```

Python3 Solution:

```
"""
Problem: Combination Sum IV
```

```
Difficulty: Medium
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def combinationSum4(self, nums: List[int], target: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def combinationSum4(self, nums, target):
"""
:type nums: List[int]
:type target: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Combination Sum IV
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} nums
* @param {number} target
* @return {number}
*/
var combinationSum4 = function(nums, target) {
};
```

TypeScript Solution:

```
/**
* Problem: Combination Sum IV
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function combinationSum4(nums: number[], target: number): number {

};
```

C# Solution:

```
/*
* Problem: Combination Sum IV
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public int CombinationSum4(int[] nums, int target) {

}
}
```

C Solution:

```
/*
* Problem: Combination Sum IV
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int combinationSum4(int* nums, int numsSize, int target) {

}
```

Go Solution:

```
// Problem: Combination Sum IV
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func combinationSum4(nums []int, target int) int {

}
```

Kotlin Solution:

```
class Solution {
fun combinationSum4(nums: IntArray, target: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func combinationSum4(_ nums: [Int], _ target: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Combination Sum IV
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn combination_sum4(nums: Vec<i32>, target: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} target
# @return {Integer}
def combination_sum4(nums, target)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $target
* @return Integer
*/
function combinationSum4($nums, $target) {

}
}
```

Dart Solution:

```
class Solution {
int combinationSum4(List<int> nums, int target) {

}
}
```

Scala Solution:

```
object Solution {
def combinationSum4(nums: Array[Int], target: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec combination_sum4(nums :: [integer], target :: integer) :: integer
def combination_sum4(nums, target) do

end
end
```

Erlang Solution:

```
-spec combination_sum4(Nums :: [integer()], Target :: integer()) ->
integer().
combination_sum4(Nums, Target) ->
.
```

Racket Solution:

```
(define/contract (combination-sum4 nums target)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```