Problem 35: Search Insert Position

Problem Information

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

Problem Description

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with

O(log n)

runtime complexity.

Example 1:

Input:

nums = [1,3,5,6], target = 5

Output:

2

Example 2:

Input:

nums = [1,3,5,6], target = 2

Output:

1

Example 3:

Input:

nums = [1,3,5,6], target = 7

Output:

4

Constraints:

1 <= nums.length <= 10

4

-10

4

<= nums[i] <= 10

4

nums

contains

distinct

values sorted in

ascending

order.

-10

4

<= target <= 10

4

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
```

```
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (search-insert nums target)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Search Insert Position
```

```
* Difficulty: Easy
* Tags: array, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
};
```

Java Solution:

```
/**
* Problem: Search Insert Position
* Difficulty: Easy
* Tags: array, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Search Insert Position
Difficulty: Easy
Tags: array, sort, search

Approach: Use two pointers or sliding window technique
```

```
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Search Insert Position
* Difficulty: Easy
* Tags: array, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

TypeScript Solution:

```
/**
* Problem: Search Insert Position
* Difficulty: Easy
* Tags: array, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

C# Solution:

```
/*
* Problem: Search Insert Position
* Difficulty: Easy
* Tags: array, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

C Solution:

```
/*
* Problem: Search Insert Position
* Difficulty: Easy
* Tags: array, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

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

}
```

Go Solution:

```
// Problem: Search Insert Position
// Difficulty: Easy
// Tags: array, sort, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Search Insert Position
// Difficulty: Easy
// Tags: array, sort, search
```

```
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (search-insert nums target)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```