Problem 868: Binary Gap

Problem Information

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

Problem Description

Given a positive integer

n

, find and return

the

longest distance

between any two

adjacent

1

's in the binary representation of

n

. If there are no two adjacent

1

's, return

0

.

Two

1

's are

adjacent

if there are only

0

's separating them (possibly no

0

's). The

distance

between two

1

's is the absolute difference between their bit positions. For example, the two

1

's in

"1001"

have a distance of 3.

Example 1:

Input:

n = 22

Output:

2

Explanation:

22 in binary is "10110". The first adjacent pair of 1's is "

1

0

1

10" with a distance of 2. The second adjacent pair of 1's is "10

11

0" with a distance of 1. The answer is the largest of these two distances, which is 2. Note that
"

1

01

1

0" is not a valid pair since there is a 1 separating the two 1's underlined.

Example 2:

Input:

n = 8

Output:

0

Explanation:

8 in binary is "1000". There are not any adjacent pairs of 1's in the binary representation of 8, so we return 0.

Example 3:

Input:

n = 5

Output:

2

Explanation:

5 in binary is "101".

Constraints:

1 <= n <= 10

9

Code Snippets

C++:

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

}
```

```
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

```
int binaryGap(int n) {

}
```

Go:

```
func binaryGap(n int) int {

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (binary-gap n)
(-> exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Binary Gap
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
};
```

Java Solution:

```
/**
* Problem: Binary Gap
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
```

```
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int binaryGap(int n) {

}
}
```

Python3 Solution:

```
"""
Problem: Binary Gap
Difficulty: Easy
Tags: general

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Binary Gap
* Difficulty: Easy
```

```
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

TypeScript Solution:

```
/**
* Problem: Binary Gap
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

function binaryGap(n: number): number {

};
```

C# Solution:

```
/*
* Problem: Binary Gap
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

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

}
}
```

C Solution:

```
/*
* Problem: Binary Gap
* Difficulty: Easy
* Tags: general
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

int binaryGap(int n) {

}
```

Go Solution:

```
// Problem: Binary Gap
// Difficulty: Easy
// Tags: general
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

func binaryGap(n int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Binary Gap
// Difficulty: Easy
// Tags: general
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

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

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {
```

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (binary-gap n)
(-> exact-integer? exact-integer?)
)
```