Problem Description

The problem requires us to determine if a given positive integer n is a happy number. A happy number is one that follows a specific process: Take the original number and replace it with the sum of the squares of its digits. Continue this process repeatedly. If eventually, the number transforms into 1, then it is a happy number. If the sequence of transformations forms an endless loop that never reaches 1, then it is not a happy number. The challenge is to create an algorithm to check whether the number reaches 1 or gets trapped in a cycle that doesn't include 1.

Intuition

To solve the problem, the concept of detecting cycles in a sequence can be applied. This is a common challenge in many algorithm problems and can often be addressed using two-pointers, namely the slow pointer and the fast pointer. The intuition here is to use the Floyd's cycle detection algorithm which is efficient for detecting cycles.

The next(x) function is created to compute the sum of the squares of the digits of x. Using this function, we then iterate through the process using two pointers, slow and fast. Initially, slow is assigned to n, and fast is assigned to the next value in the sequence. During each step of the iteration:

- fast is moved two steps forward (two applications of the next function).
- If a cycle exists that does not include 1, then slow and fast will eventually meet. If they meet at 1, it is a happy number. If they meet

slow is moved one step forward (one application of the next function).

at any number other than 1, it is not a happy number. The loop continues until the slow and fast pointers are equal, and the return statement checks if the number they met at is 1, thereby confirming if n is a happy number or not.

The solution implements Floyd's cycle detection algorithm, which is an efficient way to detect cycles in a sequence. Here's a step-

Solution Approach

by-step explanation of the implementation based on the provided code: 1. A helper function named next(x) is defined inside the Solution class. This function calculates the sum of the squares of the

- digits of a number x. Inside next(x), a variable y is initialized to 0, which will hold the sum of the squares of digits of x.
 - A loop is used to process each digit of x individually, where divmod(x, 10) returns two values: the quotient x (after removing)
 - the last digit) and the remainder v (the last digit). The square of the digit (v * v) is added to y.
 - The loop continues until all digits of x are processed.
 - The function returns y, which is the calculated sum of squares of the digits of the input number x.
- 2. Inside the isHappy method, two pointers slow and fast are initialized. slow starts at the input number n, and fast starts at the

steps), effectively advancing the pointers in the sequence until they either meet or fast reaches 1.

because we arrived at 1 without being trapped in a cycle. The function returns True in this case.

- number obtained after one iteration of the next function on n. 3. The solution then uses a while loop to continue as long as slow does not equal fast. The idea is to move slow one step at a time
- and fast two steps at a time through the sequence generated by applying the next function. • The condition slow != fast ensures that the loop continues until the two pointers meet, which indicates a cycle has been
- detected. 4. In each iteration of the while loop, slow is updated to next(slow) (one step) and fast is updated to next(next(fast)) (two
- 5. Once the loop breaks (either slow == fast or fast reaches 1), the algorithm checks if slow == 1. If it is, then n is a happy number
 - If slow != 1, it means that slow and fast met at a number other than 1, which indicates a cycle not including 1. The function returns False, meaning n is not a happy number.
- By implementing this approach, cycles are detected efficiently without the need for extra storage to keep track of the numbers already encountered, which is a direct benefit of using Floyd's cycle detection algorithm.

Example Walkthrough

Let's illustrate the solution approach with an example. Consider the number n = 19. We want to determine if it's a happy number.

According to the process, we need to calculate the sum of the squares of its digits, repeat the process with the new number, and

check if we eventually reach 1 or start looping without hitting 1. 1. Start with initial number n = 19. 2. Use the next function to calculate sum of squares of digits: $1^2 + 9^2 = 1 + 81 = 82$.

- 4. Enter the loop:
- First iteration:

3. Set slow = n (19 initially) and fast = next(n) (82 initially).

- slow becomes next(19) which is 82.
 - fast becomes next(next(82)) which is next(68) \rightarrow next(100) \rightarrow 1 (since 1^2 + 0^2 + 0^2 = 1).
 - Since fast has reached 1, the loop continues, but the condition for slow to meet fast, which was skipped in this example

Define a helper function to compute the next number in the sequence

Divide x by 10, saving the remainder and the quotient

because fast hit 1 before the comparison, wasn't necessary.

- 5. The loop checks if slow == fast or if fast == 1. In this case, fast is already 1, so we can conclude that 19 is a happy number
- without slow and fast needing to meet. 6. The isHappy function would return True, confirming that 19 is indeed a happy number.
- In this example, the cycle detection wasn't necessary because we quickly reached 1. However, for larger numbers or numbers that fall into an endless loop, the cycle detection becomes a crucial part of identifying a non-happy number by observing slow and fast
- pointers meeting at some number other than 1.

total_sum = 0 # Continue until x is reduced to zero while x:

9

10

11

12

13

14

16

17

18

Python Solution

def isHappy(self, n: int) -> bool:

int fastRunner = getNext(n);

return slowRunner == 1;

return sumOfSquares;

return slowPointer == 1;

while (slowPointer !== fastPointer) {

return fastPointer === 1;

// Moves slow pointer by one step

// Moves fast pointer by two steps

slowPointer = getNextNumber(slowPointer);

while (slowPointer != fastPointer) {

// If slowPointer is 1, n is a happy number

int slowPointer = n; // Slow pointer moves one step

int fastPointer = getNextNumber(n); // Fast pointer moves two steps

// Continue moving the pointers until they meet or find a happy number

while (slowRunner != fastRunner) {

slowRunner = getNext(slowRunner);

// Loop until the two pointers meet or we find a happy number.

// If the slow runner reaches 1, then the number is happy.

fastRunner = getNext(getNext(fastRunner)); // Move fast pointer by two steps.

// If the pointers meet and it's not at 1, then a cycle is detected and the number is not happy.

x, digit = divmod(x, 10)

def get_next_number(x):

class Solution:

```
# Add the square of the remainder to total_sum
10
                   total_sum += digit * digit
11
               return total_sum
12
13
14
           # Initialize two pointers for detecting cycles (Floyd's cycle detection algorithm)
15
           slow = n
           fast = get_next_number(n)
16
17
           # Loop until the two pointers meet or we find a happy number
18
           while slow != fast:
               # The slow pointer moves one step at a time
20
               slow = get_next_number(slow)
21
22
               # The fast pointer moves two steps at a time
23
               fast = get_next_number(get_next_number(fast))
24
           # The number is happy if and only if the loop ends with slow equals to 1
25
26
           return slow == 1
27
Java Solution
   class Solution {
       // Method to determine if a number is a happy number.
       public boolean isHappy(int n) {
           // Initialize slow and fast pointers to detect cycle.
           int slowRunner = n;
```

// Move slow pointer by one step.

```
19
       // Helper method to calculate the next number in the sequence.
20
       private int getNext(int number) {
21
22
           int sumOfSquares = 0;
23
           while (number > 0) {
               int digit = number % 10;
24
                                                      // Extract the last digit of the current number.
               sumOfSquares += digit * digit;
25
                                                      // Add the square of the extracted digit to the sum.
26
               number /= 10;
                                                       // Remove the last digit from the current number.
28
           return sumOfSquares;
29
30 }
31
C++ Solution
 1 #include<cmath> // Required for pow function
   // Solution class to determine if a number is a 'happy number'
   class Solution {
   public:
       // Function to check if a number is happy
       bool isHappy(int n) {
           // Lambda function to calculate the next number by summing the squares of the digits
           auto getNextNumber = [](int currentNumber) {
 9
               int sumOfSquares = 0;
10
               while (currentNumber > 0) {
11
                   int digit = currentNumber % 10; // Get last digit
12
13
                   sumOfSquares += std::pow(digit, 2); // Add square of the digit to sum
                   currentNumber /= 10; // Remove the last digit
14
```

// Initialize two pointers for the cycle detection (Floyd's Tortoise and Hare algorithm)

```
25
                slowPointer = getNextNumber(slowPointer); // Move slow pointer by one step
26
                fastPointer = getNextNumber(getNextNumber(fastPointer)); // Move fast pointer by two steps
27
28
```

15

16

17

18

19

20

21

22

23

24

29

30

31

20

21

22

23

24

25

26

27

28

29

32 }; 33 Typescript Solution 1 // Determines if a number is a "happy number" 2 // A happy number is a number in which the sum of the square of digits eventually reaches 1 // If it enters a cycle without reaching 1, it is not a happy number function isHappy(n: number): boolean { // This function calculates the sum of the squares of the digits of a given number const getNextNumber = (currentNumber: number): number => { let sumOfSquares = 0; while (currentNumber !== 0) { let digit = currentNumber % 10; sumOfSquares += digit ** 2; 10 currentNumber = Math.floor(currentNumber / 10); 11 12 13 return sumOfSquares; 14 **}**; 15 // Initializes two pointers for detecting cycles 16 17 let slowPointer = n; let fastPointer = getNextNumber(n); 18

// Uses Floyd's cycle detection algorithm to determine if a cycle exists

// If fastPointer equals 1, it means we've reached the happy number condition

fastPointer = getNextNumber(getNextNumber(fastPointer));

constant space as well, as it simply computes the next value in the sequence.

The given Python code uses the Floyd's cycle detection algorithm (also known as the tortoise and the hare algorithm) to determine if a number is a "happy number". The functions involve iterating over the digits of the number, squaring them, and summing them until

The time complexity of the algorithm is a bit tricky to analyze because it depends on understanding how many steps it takes before

we find a cycle or reach the number 1. Experimental results suggest that we will either reach 1 or fall into a cycle after a number of

steps that is at most linear in the number of digits of the original number n. Therefore, for practical purposes, we often assume the

time complexity to be O(log n) where n is the input number, as the sequence of transformations will be at most O(log n) before

Time and Space Complexity the process repeats a sequence or arrives at 1, which implies a happy number.

30 } 31

repeating or reaching 1. The space complexity of the algorithm is 0(1). This is because the algorithm only uses a few variables to store the slow and fast pointers (slow and fast) and does not allocate any additional space that grows with the input n. The helper function next(x) uses