Solution: Happy Number

Let's solve the Happy Number problem using the Fast and slow pointers pattern.

We'll cover the following

Statement

Write an algorithm to determine if a number n is a happy number.

We use the following process to check if a given number is a happy number:

- Starting with the given number n, replace the number with the sum of the squares of its digits.
- Repeat the process until:
  - The number equals 1, which will depict that the given number n is a happy number.
  - It enters a cycle, which will depict that the given number n is not a happy number.

Return TRUE if n is a happy number, and FALSE if not.

Constraints

- $1 \leq n \leq 2^{31} - 1$

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The brute force approach is to repeatedly calculate the squared sum of digits of the input number and store the computed sum in a hash set. For every calculation, we check if the sum is already present in the set. If yes, we've detected a cycle and should return FALSE. Otherwise, we add it to our hash set and continue further. If our sum converges to 1, we've found a happy number.

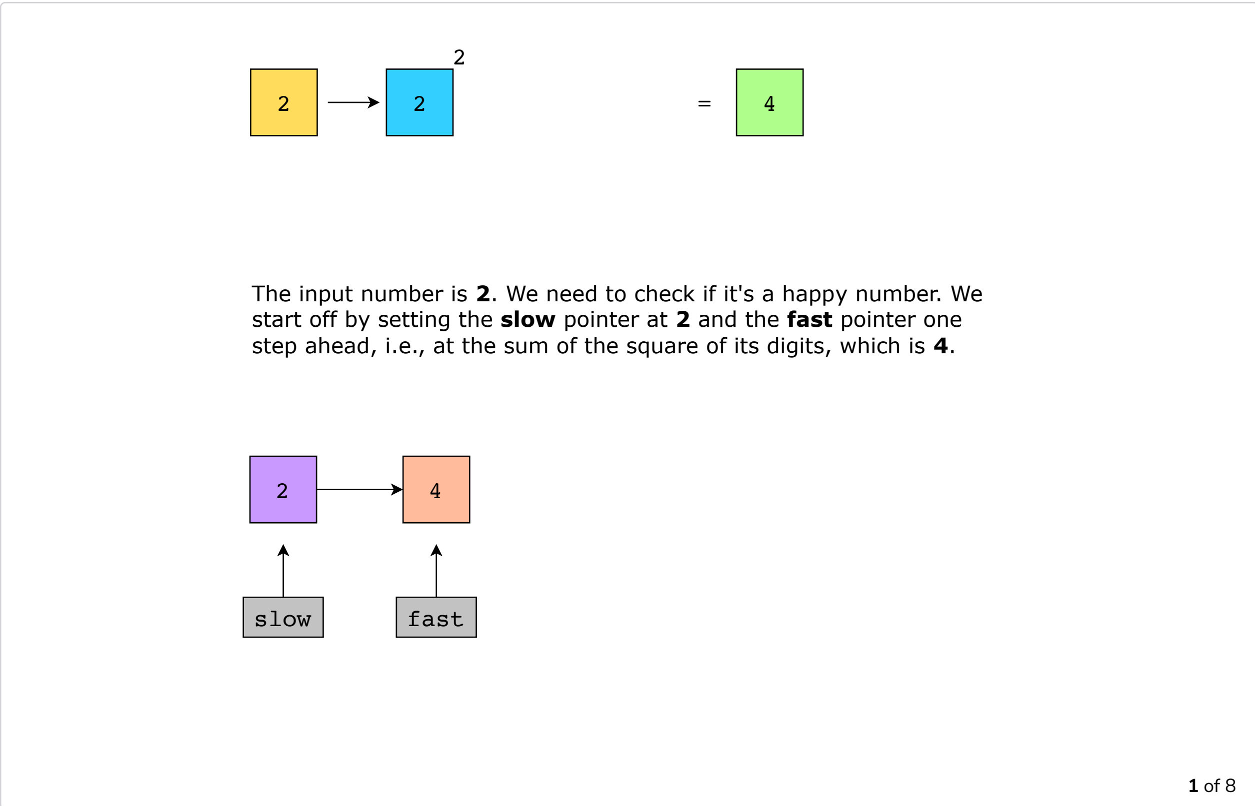
While this approach works well for small numbers, we might have to perform several computations for larger numbers to get the required result. So, it might get infeasible for such cases. The time complexity of this approach is $O(\log n)$. The space complexity is $O(\log n)$ since we're using additional space to store our calculated sums.

Optimized approach using Fast and Slow Pointers pattern

An efficient approach to solve this problem is to use fast and slow pointers. We know that a unhappy number eventually gets stuck in an infinite loop. However, there is no way for our program to detect this loop and terminate, unless we keep track of the calculated sums, which requires additional space.

If we use the fast and slow pointers approach here, the fast pointer would eventually reach 1, in which case we will return TRUE. Otherwise, it would meet the slow pointer, which would mean that the two pointers are in an endless loop, and we can return FALSE.

As an example, suppose we have the number 2 as our n. This is what the infinite loop would look like:



> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

We will start off our solution by constructing a helper function to calculate the squared sum of digits of the input number. We know that we need to isolate the digits in our number to calculate the squared sum. This can be achieved by repeatedly removing the last digit of the number and adding its squared value to the total sum.

The helper function will find the last digit of the given number by taking its modulus with 10. We'll store this in a variable `digit`. Now, since we've already separated the last digit, we can get the remaining digits by dividing the number by 10. Lastly, we'll store the squared sum of `digit` in a variable named `totalSum`. We'll repeat this until our number becomes 0.

To understand this better, consider a number, 19:

First iteration

- `digit` $= 19\%10 = 9$ (last digit)
- `number` $= 19/10 = 1$ (remaining digit(s))
- `totalSum` $= 9^2 = 81$

Second iteration

- `digit` $= 1\%10 = 1$ (last digit)
- `number` $= 1/10 = 0$ (remaining digit(s))
- `totalSum` $= 81 + 1^2 = 82$

As the `number` has become 0, we'll terminate our program here. The squared sum of the digits in 19 is 82.

Java

```java
class HappyNumber {

  public static String printStringWithMarkers(String strn, int pValue) {
      String out = "";
      for (int i = 0; i < pValue; i++) {
        out += String.valueOf(strn.charAt(i));
      }
      out += "«";
      out += String.valueOf(strn.charAt(pValue)) + "»";
      for (int i = pValue + 1; i < strn.length(); i++) {
        out += String.valueOf(strn.charAt(i));
      }
      return out;
  }
  public static int sumOfSquaredDigits(int number) {
      int totalSum = 0;
      System.out.println("\tCalculating the sum of squared digits");
      System.out.println("\tTotal sum: " + totalSum);
      int i = 1;
      while (number > 0) {
        System.out.println("\tLoop iteration: " + i);
        int a = String.valueOf(number).length();
        System.out.println("\t\tNumber:  " + number);
        int digit = number % 10;
        System.out.println("\t\tWe will start with the last digit of the number " + digit);
        System.out.println("\t\t" + printStringWithMarkers(String.valueOf(number), a - 1) + " → Last Dig
        System.out.println("\t\tUpdating number → number/10 = " + number + "/10");
        number = number / 10;
```

Happy Number

Next, we'll initialise two variables `fast` and `slow` with the input number, and the sum of its digits respectively. In each iteration, we'll move `slow` one step forward and `fast` two steps forward. That is, we'll call the `sumOfSquaredDigits()` function once for `slow` and twice for `fast`.

- `slow` $=$ `sumOfSquaredDigits(slow)`
- `fast` $=$ `sumOfSquaredDigits(sumOfSquaredDigits(fast))`

If at any instance `fast` becomes 1, we've found a happy number. We'll return TRUE in this case. Otherwise, if `fast` becomes equal to `slow`, we've found a loop and will return FALSE.

Java

```java
class HappyNumber {
```

```java

  public static int sumOfSquaredDigits(int number) {
      int totalSum = 0;
      while (number != 0) {
        int digit = number % 10;
        number = number / 10;
        totalSum += (Math.pow(digit, 2));
      }
      System.out.println("\t\tSum of squared digits: " + totalSum);
      return totalSum;
  }
  public static boolean isHappyNumber(int n) {
      int slowPointer = n; // The slow pointer value
      System.out.println("\tSetting slow pointer = input number " + slowPointer);
      System.out.println("\tSetting fast pointer = sum of squared digits of " + n);
      int fastPointer = sumOfSquaredDigits(n); // The fast pointer value
      System.out.println("\tFast pointer:" + fastPointer);
      while (fastPointer != 1 && slowPointer != fastPointer) { // Terminating condition
          System.out.println("\n\tRepeatedly updating slow and fast pointers\n");
          // Incrementing the slow pointer by 1 iteration
          slowPointer = sumOfSquaredDigits(slowPointer);
          System.out.println("\tThe updated slow pointer is " + slowPointer);
          // Incrementing the fast pointer by 2 iterations
          fastPointer = sumOfSquaredDigits(sumOfSquaredDigits(fastPointer));
          System.out.println("\tThe updated fast pointer is " + fastPointer + "\n");
      }
      System.out.println("\tIs it a happy number?: " + (fastPointer == 1)); // If 1 is found then it retu
```

Happy Number

Just the code

Here's the complete solution to this problem:

Java

```java
class HappyNumber {

  public static int sumOfSquaredDigits(int number) {
      int totalSum = 0;
      while (number != 0) {
        int digit = number % 10;
        number = number / 10;
        totalSum += (Math.pow(digit, 2));
      }
      return totalSum;
  }
  public static boolean isHappyNumber(int n) {
      int slowPointer = n;
      int fastPointer = sumOfSquaredDigits(n);

      while (fastPointer != 1 && slowPointer != fastPointer) {
          slowPointer = sumOfSquaredDigits(slowPointer);
          fastPointer =  sumOfSquaredDigits(sumOfSquaredDigits(fastPointer));
      }
      return fastPointer == 1;
  }

  public static void main(String args[]) {
    int a[] = {1, 5, 19, 25, 7};
    for (int i = 0; i < a.length; i++) {
      System.out.println((i + 1) + ".\tInput Number: " + a[i]);
      String output = isHappyNumber(a[i]) ? "True" : "False";
```

Happy Number

Solution summary

We maintain track of two values using a slow pointer and a fast pointer. The slow runner advances one number at each step, while the fast runner advances two numbers. We detect if there is any cycle by comparing the two values and checking if the fast runner has indeed reached the number one. We return True or False depending on if those conditions are met.

Time complexity

The time complexity for this algorithm is $O(\log n)$, where n is the input number.

The worst case time complexity of this algorithm is given by the case of a non-happy number, since it gets stuck in a cycle, whereas a happy number quickly converges to 1. Let's first calculate the time complexity of the **Sum Digits** function. Since we are calculating the sum of all digits in a number, the time complexity of this function is $O(\log n)$, because the number of digits in the number n is $\log_{10} n$.

1. **Numbers with three digits:** The largest three-digit number is 999. The sum of the squares of its digits is 243. Therefore, for three-digit numbers, no number in the cycle can go beyond 243. Therefore, the time complexity in this case is given as $O(243 \times 3)$, where 243 is the maximum count of numbers in a cycle and 3 is the number of digits in a three-digit number. This is why the time complexity in the case of numbers with three digits comes out to be $O(1)$.
2. **Numbers with more than three digits:** Any number with more than three digits will lose at least one digit at every step until it becomes a three-digit number. For example, the first four-digit number that we can get in the cycle is 1053, which is the sum of the square of the digits in 9999999999999. Therefore, the time complexity of any number with more than three digits can be expressed as $O(\log n + \log\log n + \log\log\log n + \ldots)$. Since $O(\log n)$ is the dominating term, we can write the time complexity as $O(\log n)$.

Therefore, the total time complexity comes out to be $O(1 + \log n)$, which is $O(\log n)$.

Space complexity

The space complexity for this algorithm is $O(1)$.

Back
Happy Number

Next
Linked List Cycle

Mark as Completed