Previous Next

202. Happy Number

Nov. 4, 2019 | 64.8K views

Average Rating: 4.91 (100 votes)

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

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1. Those numbers for which this process **ends in 1** are happy numbers.

Return True if `n` is a happy number, and False if not.

Example:

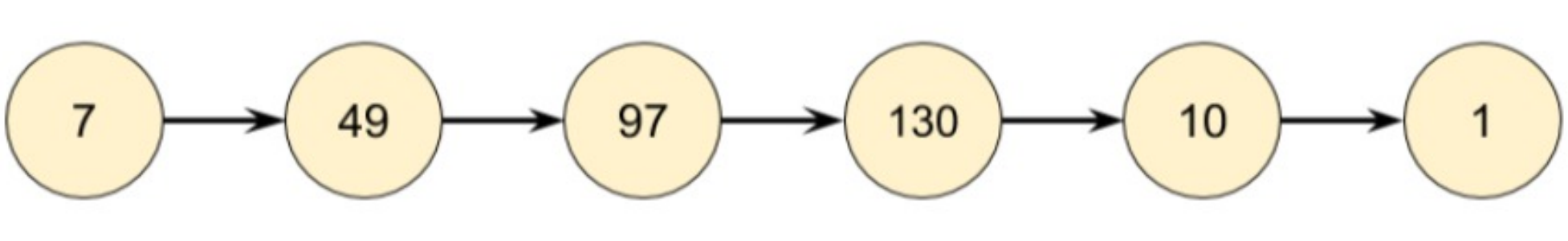
```
Input: 19
Output: true
Explanation:
1^2 + 9^2 = 82
8^2 + 2^2 = 68
6^2 + 8^2 = 100
1^2 + 0^2 + 0^2 = 1
```

Solution

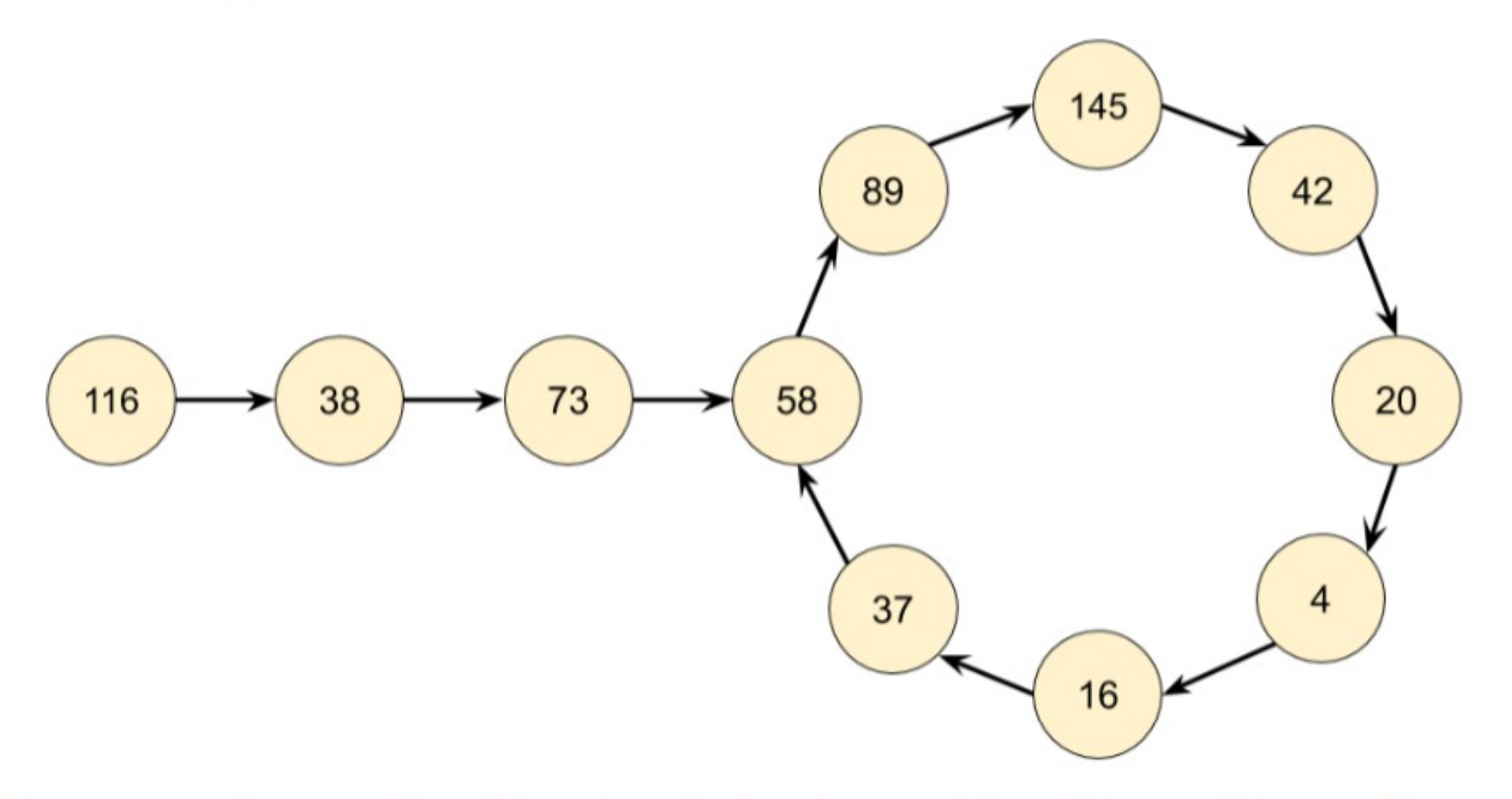
Approach 1: Detect Cycles with a HashSet

Intuition

A good way to get started with a question like this is to make a couple of examples. Let's start with the number 7. The next number will be 49 (as $7^2 = 49$), and then the next after that will be 97 (as $4^2 + 9^2 = 97$). We can continually repeat the process of squaring and then adding the digits until we get to 1. Because we got to 1, we know that 7 is a happy number, and the function should return `true`.



As another example, let's start with 116. By repeatedly applying the squaring and adding process, we eventually get to 58, and then a bit after that, we get *back to* 58. Because we are back at a number we've already seen, we know there is a cycle, and therefore it is impossible to ever reach 1. So for 116, the function should return `false`.



Based on our exploration so far, we'd expect continually following links to end in one of three ways. 1. It eventually gets to 1. 2. It eventually gets stuck in a cycle. 3. It keeps going higher and higher, up towards infinity.

That 3rd option sounds really annoying to detect and handle. How would we even know that it is going to continue going up, rather than eventually going back down, possibly to 1? Luckily, it turns out we don't need to worry about it. Think carefully about what the largest next number we could get for each number of digits is.

Digits	Largest	Next
1	9	81
2	99	162
3	999	243
4	9999	324
13	9999999999999	1053

For a number with 3 digits, it's impossible for it to ever go larger than 243. This means it will have to either get stuck in a cycle below 243 or go down to 1. Numbers with 4 or more digits will always lose a digit at each step until they are down to 3 digits. So we know that *at worst*, the algorithm might cycle around all the numbers under 243 and then go back to one it's already been to (a cycle) or go to 1. But it won't go on indefinitely, allowing us to rule out the 3rd option.

Even though you don't need to handle the 3rd case in the code, you still need to understand *why* it can never happen, so that you can justify why you didn't handle it.

Algorithm

There are 2 parts to the algorithm we'll need to design and code. 1. Given a number n, what is its *next* number? 2. Follow a chain of numbers and detect if we've entered a cycle.

Part 1 can be done by using the division and modulus operators to repeatedly take digits off the number until none remain, and then squaring each removed digit and adding them together. Have a careful look at the code for this, "picking digits off one-by-one" is a useful technique you'll use for solving a lot of different problems.

Part 2 can be done using a **HashSet**. Each time we generate the next number in the chain, we check if it's already in our HashSet. - If it is *not* in the HashSet, we should add it. - If it *is* in the HashSet, that means we're in a cycle and so should return `false`.

The reason we use a **HashSet** and *not* a Vector, List, or Array is because we're repeatedly checking whether or not numbers are in it. Checking if a number is in a HashSet takes $O(1)$ time, whereas for the other data structures it takes $O(n)$ time. Choosing the correct data structures is an essential part of solving these problems.

Java Python Copy

```java
class Solution {

    private int getNext(int n) {
        int totalSum = 0;
        while (n > 0) {
            int d = n % 10;
            n = n / 10;
            totalSum += d * d;
        }
        return totalSum;
    }

    public boolean isHappy(int n) {
        Set<Integer> seen = new HashSet<>();
        while (n != 1 && !seen.contains(n)) {
            seen.add(n);
            n = getNext(n);
        }
        return n == 1;
    }
}
```

Complexity Analysis

Determining the *time complexity* for this problem is challenging for an "easy" level question. If you're new to these problems, have a go at calculating the time complexity for *just* the `getNext(n)` function (don't worry about how many numbers will be in the chain).

- Time complexity : $O(243 \cdot 3 + \log n + \log\log n + \log\log\log n)...$ = $O(\log n)$.

  Finding the **next** value for a given number has a cost of $O(\log n)$ because we are processing each digit in the number, and the number of digits in a number is given by $\log n$.

  To work out the *total* time complexity, we'll need to think carefully about how many numbers are in the chain, and how big they are.

  We determined above that once a number is below 243, it is impossible for it to go back up above 243. Therefore, based on our very shallow analysis we know for *sure* that once a number is below 243, it is impossible for it to take more than another 243 steps to terminate. Each of these numbers has at most 3 digits. With a little more analysis, we could replace the 243 with the length of the longest number chain below 243, however because the constant doesn't matter anyway, we won't worry about it.

  For an n above 243, we need to consider the cost of each number in the chain that is above 243. With a little math, we can show that in the worst case, these costs will be $O(\log n) + O(\log\log n) + O(\log\log\log n)...$. Luckily for us, the $O(\log n)$ is the dominating part, and the others are all tiny in comparison (collectively, they add up to less than $\log n$), so we can ignore them.

- Space complexity : $O(\log n)$. Closely related to the time complexity, and is a measure of what numbers we're putting in the HashSet, and how big they are. For a large enough n, the most space will be taken by n itself.

  We can optimize to $O(243 \cdot 3) = O(1)$ easily by only saving numbers in the set that are less than 243, as we have already shown that for numbers that are higher, it's impossible to get back to them anyway.

It might seem worrying that we're simply dropping such "large" constants. But this is what we do in Big O notation, which is a measure of how long the function will take, as the *size of the input increases*.

Think about what would happen if you had a number with *1 million* digits in it. The first step of the algorithm would process those million digits, and then the next value would be, at most (pretend all the digits are 9), be $81 * 1,000,000 = 81,000,000$. In just one step, we've gone from a million digits, down to just 8. The largest possible 8 digit number we could get is $99,9999,999$, which then goes down to $81 * 8 = 648$. And then from here, the cost will be the same as if we'd started with a 3 digit number. Starting with 2 million digits (a **massively** larger number than one with a 1 million digits) would only take roughly twice as long, as again, the dominant part is summing the squares of the 2 million digits, and the rest is *tiny* in comparison.

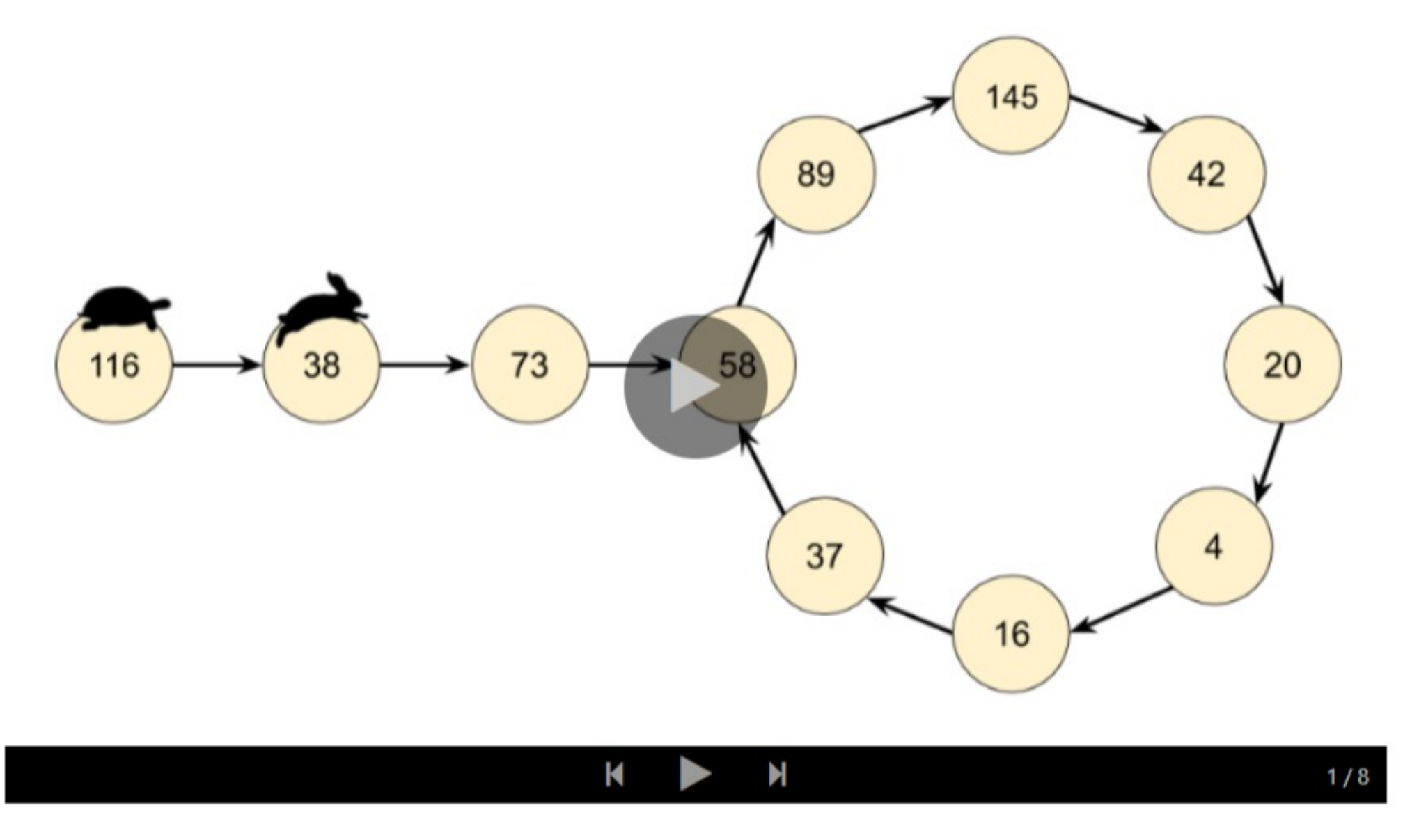
Approach 2: Floyd's Cycle-Finding Algorithm

Intuition

The chain we get by repeatedly calling `getNext(n)` is an *implicit* **LinkedList**. *Implicit* means we don't have actual LinkedNode's and pointers, but the data does still form a LinkedList structure. The starting number is the head "node" of the list, and all the other numbers in the chain are nodes. The next pointer is obtained with our `getNext(n)` function above.

Recognizing that we actually have a LinkedList, it turns out that this question is almost the same as another Leetcode problem, detecting if a linked list has a cycle. As @Freezen has pointed out), we can therefore use Floyd's Cycle-Finding Algorithm here. This algorithm is based on 2 runners running around a circular race track, a fast runner and a slow runner. In reference to a famous fable, many people call the slow runner the "tortoise" and the fast runner the "hare".

Regardless of where the tortoise and hare start in the cycle, they are guaranteed to eventually meet. This is because the hare moves one node closer to the tortoise (in their direction of movement) each step.



Algorithm

Instead of keeping track of just one value in the chain, we keep track of 2, called the slow runner and the fast runner. At each step of the algorithm, the slow runner goes forward by 1 number in the chain, and the fast runner goes forward by 2 numbers (nested calls to the `getNext(n)` function).

If n *is* a happy number, i.e. there is no cycle, then the fast runner will eventually get to 1 before the slow runner.

If n *is not* a happy number, then eventually the fast runner and the slow runner will be on the same number.

Java Python Copy

```java
class Solution {

     public int getNext(int n) {
        int totalSum = 0;
        while (n > 0) {
            int d = n % 10;
            n = n / 10;
            totalSum += d * d;
        }
        return totalSum;
    }

    public boolean isHappy(int n) {
        int slowRunner = n;
        int fastRunner = getNext(n);
        while (fastRunner != 1 && slowRunner != fastRunner) {
            slowRunner = getNext(slowRunner);
            fastRunner = getNext(getNext(fastRunner));
        }
        return fastRunner == 1;
    }
}
```

Complexity Analysis

- Time complexity : $O(\log n)$. Builds on the analysis for the previous approach, except this time we need to analyse how much extra work is done by keeping track of two places instead of one, and how many times they'll need to go around the cycle before meeting.

If there is no cycle, then the fast runner will get to 1, and the slow runner will get halfway to 1. Because there were 2 runners instead of 1, we know that at worst, the cost was $O(2 \cdot \log n) = O(\log n)$.

Like above, we're treating the length of the chain to the cycle as insignificant compared to the cost of calculating the next value for the first n. Therefore, the only thing we need to do is show that the number of times the runners go back over previously seen numbers in the chain is constant.

Once both pointers are in the cycle (which will take constant time to happen) the fast runner will get one step closer to the slow runner at each cycle. Once the fast runner is one step behind the slow runner, they'll meet on the next step. Imagine there are k numbers in the cycle. If they started at $k - 1$ places apart (which is the furthest apart they can start), then it will take $k - 1$ steps for the fast runner to reach the slow runner, which again is constant for our purposes. Therefore, the dominating operation is still calculating the next value for the starting n, which is $O(\log n)$.

- Space complexity : $O(1)$. For this approach, we don't need a HashSet to detect the cycles. The pointers require constant extra space.

Approach 3: Hardcoding the Only Cycle (Advanced)

Intuition

The previous two approaches are the ones you'd be expected to come up with in an interview. This third approach is ***not something you'd write in an interview***, but is aimed at the mathematically curious among you as it's quite interesting.

What's the biggest number that could have a next value bigger than itself? Well we know it has to be less than 243, from the analysis we did previously. Therefore, we know that any cycles must contain numbers *smaller* than 243, as anything bigger could not be cycled back to. With such small numbers, it's not difficult to write a brute force program that finds all the cycles.

If you do this, you'll find there's only *one* cycle: $4 \rightarrow 16 \rightarrow 37 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4$. All other numbers are on chains that lead into this cycle, or on chains that lead into 1.

Therefore, we can just hardcode a HashSet containing these numbers, and if we ever reach one of them, then we know we're in the cycle. There's no need to keep track of where we've been previously.

Algorithm

Java Python Copy

```java
class Solution {

    private static Set<Integer> cycleMembers =
        new HashSet<>(Arrays.asList(4, 16, 37, 58, 89, 145, 42, 20));

    public int getNext(int n) {
        int totalSum = 0;
        while (n > 0) {
            int d = n % 10;
            n = n / 10;
            totalSum += d * d;
        }
        return totalSum;
    }


    public boolean isHappy(int n) {
        while (n != 1 && !cycleMembers.contains(n)) {
            n = getNext(n);
        }
        return n == 1;
    }
}
```

Complexity Analysis

Time complexity : $O(\log n)$. Same as above.

Space complexity : $O(1)$. We are not maintaining any history of numbers we've seen. The hardcoded HashSet is of a constant size.

An Alternative Implementation

Thanks @Manky for sharing this alternative with us!

This approach was based on the idea that all numbers either end at `1` or enter the cycle `{4, 16, 37, 58, 89, 145, 42, 20}`, wrapping around it infinitely.

An alternative approach would be to recognise that all numbers will either end at `1`, or go past `4` (a member of the cycle) at some point. Therefore, instead of hardcoding the entire cycle, we can just hardcode the `4`.

Java Python Copy

```java
class Solution {

    public int getNext(int n) {
        int totalSum = 0;
        while (n > 0) {
            int d = n % 10;
            n = n / 10;
            totalSum += d * d;
        }
        return totalSum;
    }

    public boolean isHappy(int n) {
        while (n != 1 && n != 4) {
            n = getNext(n);
        }
        return n == 1;
    }
}
```

This alternative has the same time and space complexity as approach 3, from a big-oh point of view. The time taken in practice for this alternative will be slower by a *constant* amount though. If the cycle was entered at 16, then the algorithm will traverse the entire cycle before getting back to 4. The space complexity will be *less* by a *constant* amount, because we're now only hardcoding `4` and not the other 7 numbers in the cycle.

Rate this article: ★★★★★

Previous Next

Comments: 33 Sort By

Type comment here... (Markdown is supported)

Preview Post

Shwetha_Anand ★28 November 13, 2019 2:03 AM

Thank you for the article, truly helped

25 Share Reply

SHOW 1 REPLY

undefitied ★92 February 2, 2020 3:33 PM

The hardcoded cycle is LOL. :)

17 Share Reply

huntz256 ★12 February 20, 2020 3:50 AM

"For an n above 243, we need to consider the cost of each number in the chain that is above 243. With a little math, we can show that in the worst case, these costs will be O(log n) + O(log log n) + O(log log log n)..." Why are these costs O(log n) + O(log log n) + O(log log log n) + ...?

11 Share Reply

SHOW 6 REPLIES

RubaParv ★11 November 5, 2019 10:03 PM

Wonderful explanations!

11 Share Reply

voquanghoa ★15 November 5, 2019 8:25 PM

Good article. Now I know the Floyd's Cycle-Finding algorithm. Thanks

10 Share Reply

uncleiroh ★36 April 2, 2020 8:54 PM

great explanation! For approach 1, I will add a little bit of my own explanation because it may help a little. For the time complexity, the way we reach O(log N) is as so: first we follow this link (https://stackoverflow.com/a/50262470) to understand why summing the digits of a number is O(log N), where N is the number itself. So for our time complexity analysis, we will take this fact for granted (that the getNext() method is O(logN). The time complexity analysis is broken up into two steps:

Read More

6 Share Reply

SHOW 1 REPLY

280045830 ★4 February 9, 2020 10:11 AM Report

Dame , only 1 & 7 lower than 10 will get to 1 eventually: why not hard code like this ?

class Solution {
public boolean isHappy(int n) {

Read More

4 Share Reply

SHOW 1 REPLY

abascus ★23 November 7, 2019 8:11 PM Report

Brilliant article, man. The level of the details and the analysis are so good.

5 Share Reply

gauravsin ★41 December 17, 2019 5:32 PM

@Hai_dee , beautifully written.

3 Share Reply

troywang6666 ★13 November 27, 2019 5:36 PM

I hope all the solutions are as concise as this one.....

3 Share Reply

‹ 1 2 3 4 ›