507. Perfect Number

March 27, 2017 | 29.1K views

Previous Next

We define the Perfect Number is a **positive** integer that is equal to the sum of all its **positive** divisors except itself.

Now, given an **integer** *n*, write a function that returns true when it is a perfect number and false when it is not.

Example:

```
Input: 28
Output: True
Explanation: 28 = 1 + 2 + 4 + 7 + 14
```

Note: The input number **n** will not exceed 100,000,000. (1e8)

Solution

Approach #1 Brute Force [Time Limit Exceeded]

Algorithm

In brute force approach, we consider every possible number to be a divisor of the given number num, by iterating over all the numbers lesser than num. Then, we add up all the factors to check if the given number satisfies the Perfect Number property. This approach obviously fails if the number num is very large.

Java

```
public class Solution {
    public boolean checkPerfectNumber(int num) {
        if (num <= 0) {
            return false;
        }
        int sum = 0;
        for (int i = 1; i < num; i++) {
            if (num % i == 0) {
                sum += i;
            }

        }
        return sum == num;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. We iterate over all the numbers lesser than n.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #2 Better Brute Force [Time Limit Exceeded]

Algorithm

We can little optimize the brute force by breaking the loop when the value of sum increase the value of num. In that case, we can directly return $false$.

Java

```
public class Solution {
    public boolean checkPerfectNumber(int num) {
        if (num <= 0) {
            return false;
        }
        int sum = 0;
        for (int i = 1; i < num; i++) {
            if (num % i == 0) {
                sum += i;
            }
            if(sum>num) {
                return false;
            }
        }
        return sum == num;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. In worst case, we iterate over all the numbers lesser than n.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #3 Optimal Solution [Accepted]

Algorithm

In this method, instead of iterating over all the integers to find the factors of num, we only iterate upto the $\sqrt{n}$. The reasoning behind this can be understood as follows.

Consider the given number num which can have m distinct factors, namely $n_1, n_2, ..., n_m$. Now, since the number num is divisible by n_i, it is also divisible by $n_j = num/n_i$ i.e. $n_i * n_j = num$. Also, the largest number in such a pair can only be up to $\sqrt{num}$ (because $\sqrt{num} \times \sqrt{num} = num$). Thus, we can get a significant reduction in the run-time by iterating only upto $\sqrt{num}$ and considering such n_i's and n_j's in a single pass directly.

Further, if $\sqrt{num}$ is also a factor, we have to consider the factor only once while checking for the perfect number property.

We sum up all such factors and check if the given number is a Perfect Number or not. Another point to be observed is that while considering 1 as such a factor, num will also be considered as the other factor. Thus, we need to subtract num from the sum.

Java

```
    public boolean checkPerfectNumber(int num) {
        if (num <= 0) {
            return false;
        }
        int sum = 0;
        for (int i = 1; i * i <= num; i++) {
            if (num % i == 0) {
                sum += i;
                if (i * i != num) {
                    sum += num / i;
                }

            }
        }
        return sum - num == num;
    }
}
```

Complexity Analysis

- Time complexity : $O(\sqrt{n})$. We iterate only over the range $1 < i \leq \sqrt{num}$.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #4 Euclid-Euler Theorem [Accepted]

Algorithm

Euclid proved that $2^{p-1}(2^p-1)$ is an even perfect number whenever 2^p-1 is prime, where p is prime.

For example, the first four perfect numbers are generated by the formula $2^{p-1}(2^p-1)$, with p a prime number, as follows:

```
for p = 2:  21(22 - 1) = 6
for p = 3:  22(23 - 1) = 28
for p = 5:  24(25 - 1) = 496
for p = 7:  26(27 - 1) = 8128.
```

Prime numbers of the form 2^p-1 are known as Mersenne primes. For 2^p-1 to be prime, it is necessary that p itself be prime. However, not all numbers of the form 2^p-1 with a prime p are prime; for example, $2^{11}-1 = 2047 = 23 \times 89$ is not a prime number.

You can see that for small value of p, its related perfect number goes very high. So, we need to evaluate perfect numbers for some primes $(2, 3, 5, 7, 13, 17, 19, 31)$ only, as for bigger prime its perfect number will not fit in 64 bits.

Java

```
public class Solution {
    public int pm(int p) {
        return (1 << (p - 1)) * ((1 << p) - 1);
    }
    public boolean checkPerfectNumber(int num) {
        int[] primes=new int[]{2,3,5,7,13,17,19,31};
        for (int prime: primes) {
            if (pm(prime) == num)
                return true;
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(\log n)$. Number of primes will be in order $\log num$.
- Space complexity : $O(\log n)$. Space used to store primes.

Rate this article:

Previous Next

Comments: 21

Sort By

Type comment here... (Markdown is supported)

Preview Post

He110world ★ 79 September 8, 2018 8:10 AM

Simple is the best.

```
class Solution:
    def checkPerfectNumber(self, num):
        """
        :type num: int
        :rtype: bool
        """
        return num in [6, 28, 496, 8128, 33550336]
```

Read More

72 Share Reply

SHOW 6 REPLIES

forest0 ★ 5 July 6, 2018 10:09 AM

Thanks for your analysis, it's a nice work.

At the very beginning when I saw your Approach 4.
I was wandering what about the odd perfect numbers.

Read More

8 Share Reply

nileshdk ★ 4 December 4, 2017 2:34 AM

You could simply start with sum as 1 start from i=2 and avoid the last substraction.

```
    public static boolean checkPerfectNumber(int num) {
        if (num == 1) {
            return false;
        }
```

Read More

4 Share Reply

saramissse ★ 6 April 11, 2017 12:10 AM

Approach #3 Better Brute Force but **[Accepted]** :)

```
bool checkPerfectNumber(int num) {
        if (num <= 1) {
            return false;
        }
```

Read More

6 Share Reply

gmotzespina ★ 2 February 15, 2020 8:10 AM

Another simple solution using Python3:

```
class Solution:
    def checkPerfectNumber(self, num: int) -> bool:
        return num in [6, 28, 496, 8128, 33550336]
```

Read More

1 Share Reply

Jianchao-li ★ 14205 January 18, 2018 7:33 PM

31 should not be included in `primes`.

0 Share Reply

SHOW 1 REPLY

tiyiyoyi ★ 4 August 4, 2019 2:33 PM

So what are these numbers used for?

0 Share Reply

trizen ★ 1 July 29, 2018 7:41 PM

O(log(n)) algorithm for arbitrary large inputs in Ruby, using the Lucas-Lehmer test for testing the primality of Mersenne numbers:

```
def lucas_lehmer(p)
```

Read More

0 Share Reply

el-khalil ★ 0 July 8, 2018 8:42 AM

```
def checkPerfectNumber(self, num):
        if num <= 0:
            return False
        factors = list(sorted(x for fun in (f1, num//f1 for f in range(1, int(num**
```

Read More

0 Share Reply

rohan1239 ★ 159 February 11, 2018 4:27 PM

public boolean checkPerfectNumber(int num) {

```
        if (num <= 1)
            return false;
```

Read More

0 Share Reply

1 2 3