Previous Next

326. Power of Three

April 19, 2016 | 132K views

Average Rating: 4.80 (131 votes)

Given an integer, write a function to determine if it is a power of three.

Example 1:

```
Input: 27
Output: true
```

Example 2:

```
Input: 0
Output: false
```

Example 3:

```
Input: 9
Output: true
```

Example 4:

```
Input: 45
Output: false
```

Follow up:
Could you do it without using any loop / recursion?

Solution

In this article we will look into ways of speeding up simple computations and why that is useful in practice.

Approach 1: Loop Iteration

One simple way of finding out if a number `n` is a power of a number `b` is to keep dividing `n` by `b` as long as the remainder is **0**. This is because we can write

$$n = b^x$$
$$n = b \times b \times \ldots \times b$$

Hence it should be possible to divide `n` by `b` `x` times, every time with a remainder of **0** and the end result to be **1**.

Java

```java
public class Solution {
    public boolean isPowerOfThree(int n) {
        if (n < 1) {
            return false;
        }

        while (n % 3 == 0) {
            n /= 3;
        }

        return n == 1;
    }
}
```

Notice that we need a guard to check that `n != 0`, otherwise the while loop will never finish. For negative numbers, the algorithm does not make sense, so we will include this guard as well.

Complexity Analysis

- Time complexity : $O(\log_b(n))$. In our case that is $O(\log_3 n)$. The number of divisions is given by that logarithm.
- Space complexity : $O(1)$. We are not using any additional memory.

Approach 2: Base Conversion

In Base 10, all powers of 10 start with the digit **1** and then are followed only by **0** (e.g. 10, 100, 1000). This is true for other bases and their respective powers. For instance in *base 2*, the representations of 10_2, 100_2 and 1000_2 are 2_{10}, 4_{10} and 8_{10} respectively. Therefore if we convert our number to base 3 and the representation is of the form 100...0, then the number is a power of 3.

Proof

Given the base 3 representation of a number as the array `s`, with the least significant digit on index 0, the formula for converting from base **3** to base **10** is:

$$\sum_{i=0}^{len(s)-1} s[i] * 3^i$$

Therefore, having just one digit of **1** and everything else **0** means the number is a power of 3.

Implementation

All we need to do is convert [4] the number to *base 3* and check if it is written as a leading **1** followed by all **0**.

A couple of built-in Java functions will help us along the way.

Java

```java
String baseChange = Integer.toString(number, base);
```

The code above converts `number` into base `base` and returns the result as a `String`. For example, `Integer.toString(5, 2) == "101"` and `Integer.toString(5, 3) == "12"`.

Java

```java
boolean matches = myString.matches("123");
```

The code above checks if a certain **Regular Expression** [2] pattern exists inside a string. For instance the above will return true if the substring "123" exists inside the string `myString`.

Java

```java
boolean powerOfThree = baseChange.matches("^10*$")
```

We will use the regular expression above for checking if the string starts with **1** `^1`, is followed by zero or more **0**s `0*` and contains nothing else `$`.

Java

```java
public class Solution {
    public boolean isPowerOfThree(int n) {
        return Integer.toString(n, 3).matches("^10*$");
    }
}
```

Complexity Analysis

- Time complexity : $O(\log_3 n)$.

  Assumptions:

  - `Integer.toString()` - Base conversion is generally implemented as a repeated division. The complexity of should be similar to our Approach 1: $O(\log_3 n)$.
  - `String.matches()` - Method iterates over the entire string. The number of digits in the base 3 representation of `n` is $O(\log_3 n)$.
- Space complexity : $O(\log_3 n)$.

  We are using two additional variables,

  - The string of the base 3 representation of the number (size $\log_3 n$)
  - The string of the regular expression (constant size)

Approach 3: Mathematics

We can use mathematics as follows

$$n = 3^i$$
$$i = \log_3(n)$$
$$i = \frac{\log_b(n)}{\log_b(3)}$$

`n` is a power of three if and only if `i` is an integer. In Java, we check if a number is an integer by taking the decimal part (using `% 1`) and checking if it is 0.

Java

```java
public class Solution {
    public boolean isPowerOfThree(int n) {
        return (Math.log10(n) / Math.log10(3)) % 1 == 0;
    }
}
```

Common pitfalls

This solution is problematic because we start using `double`s, which means we are subject to precision errors. This means, we should never use `==` when comparing `double`s. That is because the result of `Math.log10(n) / Math.log10(3)` could be `5.0000001` or `4.9999999`. This effect can be observed by using the function `Math.log()` instead of `Math.log10()`.

In order to fix that, we need to compare the result against an `epsilon`.

Java

```java
return (Math.log(n) / Math.log(3) + epsilon) % 1 <= 2 * epsilon;
```

Complexity Analysis

- Time complexity : $Unknown$ The expensive operation here is `Math.log`, which upper bounds the time complexity of our algorithm. The implementation is dependent on the language we are using and the compiler [3]
- Space complexity : $O(1)$. We are not using any additional memory. The `epsilon` variable can be inlined.

Approach 4: Integer Limitations

An important piece of information can be deduced from the function signature

In particular, `n` is of type `int`. In Java, this means it is a 4 byte, signed integer [ref]. The maximum value of this data type is **2147483647**. Three ways of calculating this value are

- Google
- `System.out.println(Integer.MAX_VALUE);`
- MaxInt = $\frac{2^{32}}{2} - 1$ since we use 32 bits to represent the number, half of the range is used for negative numbers and 0 is part of the positive numbers

Knowing the limitation of `n`, we can now deduce that the maximum value of `n` that is also a power of three is **1162261467**. We calculate this as:

$$3^{\lfloor \log_3 MaxInt \rfloor} = 3^{\lfloor 19.56 \rfloor} = 3^{19} = 1162261467$$

Therefore, the possible values of `n` where we should return `true` are 3^0, 3^1 ... 3^{19}. Since 3 is a prime number, the only divisors of 3^{19} are 3^0, 3^1 ... 3^{19}, therefore all we need to do is divide 3^{19} by `n`. A remainder of **0** means `n` is a divisor of 3^{19} and therefore a power of three.

Java

```java
public class Solution {
    public boolean isPowerOfThree(int n) {
        return n > 0 && 1162261467 % n == 0;
    }
}
```

Complexity Analysis

- Time complexity : $O(1)$. We are only doing one operation.
- Space complexity : $O(1)$. We are not using any additional memory.

Performance Measurements

Single runs of the function make it is hard to accurately measure the difference of the two solutions. On LeetCode, on the *Accepted Solutions Runtime Distribution* page, all solutions being between `15 ms` and `20 ms`. For completeness, we have proposed the following benchmark to see how the two solutions differ.

Java Benchmark Code

Java

```java
public static void main(String[] args) {
    Solution sol = new Solution();
    int iterations = 1; // See table header for this value
    for (int i = 0; i < iterations; i++) {
        sol.isPowerOfThree(i);
    }
}
```

In the table below, the values are in seconds.

Iterations	10^6	10^7	10^8	10^9	$Maxint$
Java Approach 1: (Naive)	0.04	0.07	0.30	2.47	5.26
Java Approach 2: (Strings)	0.68	4.02	38.90	409.16	893.89
Java Approach 3: (Logarithms)	0.09	0.50	4.59	45.53	97.50
Java Approach 4: (Fast)	0.04	0.06	0.08	0.41	0.78

As we can see, for small values of N, the difference is not noticeable, but as we do more iterations and the values of `n` passed to `isPowerOfThree()` grow, we see significant boosts in performance for Approach 4.

Conclusion

Simple optimizations like this might seem negligible, but historically, when computation power was an issue, it allowed certain computer programs (such as Quake 3 [1]) possible.

References

Analysis written by: @aicioara

1. https://en.wikipedia.org/wiki/Fast_inverse_square_root
2. https://en.wikipedia.org/wiki/Regular_expression
3. http://developer.classpath.org/doc/java/lang/StrictMath-source.html
4. http://www.cut-the-knot.org/recurrence/conversion.shtml