Previous Next

172. Factorial Trailing Zeroes

March 20, 2020 | 3.9K views

Given an integer *n*, return the number of trailing zeroes in *n*!.

Example 1:

```
Input: 3
Output: 0
Explanation: 3! = 6, no trailing zero.
```

Example 2:

```
Input: 5
Output: 1
Explanation: 5! = 120, one trailing zero.
```

Note: Your solution should be in logarithmic time complexity.

Solution

Approach 1: Compute the Factorial

Intuition

This approach is too slow, but is a good starting point. You wouldn't implement it in an interview, although you might briefly describe it as a possible way of solving the problem.

The simplest way of solving this problem would be to compute $n!$ and then count the number of zeroes on the end of it. Recall that factorials are calculated by multiplying all the numbers between 1 and n. For example, $10! = 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 3,628,800$. Therefore, factorials can be calculated iteratively using the following algorithm.

```
define function factorial(n):
    n_factorial = 1
    for i from 1 to n (inclusive):
        n_factorial = n_factorial * i
    return n_factorial
```

Recall that if a number has a zero on the end of it, then it is divisible by 10. Dividing by 10 will remove that zero, and shift all the other digits to the right by one place. We can, therefore, count the number of zeroes by repeatedly checking if the number is divisible by 10, and if it is then dividing it by 10. The number of divisions we're able to do is equal to the number of 0's on the end of it. This is the algorithm to count the zeroes (assuming $x \geq 1$, which is fine for this problem, as factorials are always positive integers).

```
define function zero_count(x):
    zero_count = 0
    while x is divisible by 10:
        zero_count += 1
        x = x / 10
    return zero_count
```

By putting these two functions together, we can count the number of zeroes on the end of $n!$.

Algorithm

For Java, we need to use BigInteger, because $n!$ won't fit into a `long` for even moderately small values of n.

Java Python Copy

```python
def trailingZeroes(self, n: int) -> int:

    # Calculate n!
    n_factorial = 1
    for i in range(2, n + 1):
        n_factorial *= i

    # Count how many 0's are on the end.
    zero_count = 0
    while n_factorial % 10 == 0:
        zero_count += 1
        n_factorial //= 10

    return zero_count
```

Complexity Analysis

The math involved here is very advanced, however we don't need to be precise. We can get a reasonable approximation with a little mathematical reasoning. An interviewer probably won't expect you to calculate it exactly, or even derive the approximation as carefully as we have here. However, they *might* expect you to at least have some ideas and at least attempt to reason about it. Of course, if you've claimed to be an *expert* in algorithms analysis on your résumé/CV, then they might expect you to derive the entire thing! Our main reason for including it here is because it is a nice example of working with an algorithm that is mathematically challenging to analyse.

Let n be the number we're taking the factorial of.

- Time complexity : Worse than $O(n^2)$.

  Computing a factorial is repeated multiplication. Generally when we know multiplications are on numbers within a fixed bit-size (e.g. 32 bit or 64 bit ints), we treat them as $O(1)$ operations. However, we can't do that here because the size of the numbers to be multiplied grow with the size of n.

  So, the first step here is to think about what the cost of multiplication might be, given that we can't assume it's $O(1)$. A popular way people multiply two large numbers, that you probably learned in school, has a cost of $O((\log x) \cdot (\log y))$. We'll use that in our approximation.

  Next, let's think about what multiplications we do when calculating $n!$. The first few multiplications would be as follows:

  $1 \cdot 2 = 2$
  $2 \cdot 3 = 6$
  $6 \cdot 4 = 24$
  $24 \cdot 5 = 120$
  $120 \cdot 6 = 720$
  ...

  In terms of cost, these multiplications would have costs of:

  $\log 1 \cdot \log 2$
  $\log 2 \cdot \log 3$
  $\log 6 \cdot \log 4$
  $\log 24 \cdot \log 5$
  $\log 120 \cdot \log 6$
  ...

  Recognising that the first column are all logs of factorials, we can rewrite it as follows:

  $\log 1! \cdot \log 2$
  $\log 2! \cdot \log 3$
  $\log 3! \cdot \log 4$
  $\log 4! \cdot \log 5$
  $\log 5! \cdot \log 6$
  ...

  See the pattern? Each line is of the form $(\log k!) \cdot (\log k + 1)$. What would the *last* line be? Well, the last step in calculating a factorial is to multiply by n. Therefore, the last line must be:

  $\log((n-1)!) \cdot \log(n)$

  Because we're doing each of these multiplications one-by-one, we should *add* them to get the total time complexity. This gives us:

  $\log 1! \cdot \log 2 + \log 2! \cdot \log 3 + \log 3! \cdot \log 4 + \cdots + \log((n-2)!) \cdot \log(n-1) + \log((n-1)!) \cdot \log n$

  This sequence is quite complicated to add up; instead of trying to find an exact answer, we're going to now focus on a rough *lower bound* approximation by throwing away the less significant terms. While this is not something you'd do if we needed to find the exact time complexity, it will allow us to quickly see that the time complexity is "too big" for our purposes. Often finding lower (and upper) bounds is enough to decide whether or not an algorithm is worth using. This includes in interviews!

  At this point, you'll ideally realise that the algorithm is *worse* than $O(n)$, as we're adding n terms. Given that the question asked us to come up with an algorithm that's no worse than $O(\log n)$, this is definitely not good enough. We're going to explore it a little further, but if you've understood up to this point, you're doing really well! The rest is optional.

  Continuing on, notice that $\log((n-1)!)$ is "a lot bigger" than $\log n$. Therefore, we'll just drop all these parts, leaving the logs of factorials. This gives us:

  $\log 1! + \log 2! + \log, 3! + \cdots + \log((n-2)!) + \log((n-1)!)$

  The next part involves a log rule that you might or might not have heard of. It's definitely worth remembering if you haven't heard of it though, as it can be very useful.

  $O(\log n!) = O(n \log n)$

  So, let's rewrite the sequence using this rule.

  $1 \cdot \log 1 + 2 \cdot \log 2 + 3 \cdot \log 3 + \cdots + (n-2) \cdot \log(n-2) + (n-1) \cdot \log(n-1)$

  Like before, we'll just drop the "small" $\log$ terms, and see what we're left with.

  $1 + 2 + 3 + ... + (n-2) + (n-1)$

  This is a very familiar sequence, that you should be familiar with—it describes a cost of $O(n^2)$.

  So, what can we conclude? Well, all the discarding of terms leaves us with a time complexity *less* than the real one. In other words, this factorial algorithm must be *slower* than $O(n^2)$.

  $O(n^2)$ is *definitely* not good enough!

  While this technique of throwing away terms here and there might seem a bit strange, it's very useful to make early decisions quickly, without needing to mess around with advanced math. Only once we had decided we were interested in looking at the algorithm further, would we try to come up with a more exact time complexity. And in this case, our lower bound was enough to convince us that it definitely isn't worth looking at!

  The second part, counting the zeroes at the end, is insignificant compared to the first part. There are $O(\log n!) = O(n \log n)$ digits, which is smaller than $O(n^2)$. Not to mention, only a few of them will be zeroes!

- Space complexity : $O(\log n!) = O(n \log n)$.

  In order to store $n!$, we need $O(\log n!)$ bits. As we saw above, this is the same as $O(n \log n)$.

Approach 2: Counting Factors of 5

Intuition

This approach is also too slow, however it's a likely step in the problem solving process for coming up with a logarithmic approach.

Instead of computing the factorial like in Approach 1, we can instead recognize that each 0 on the end of the factorial represents a multiplication by 10.

So, how many times do we multiply by 10 while calculating $n!$? Well, to multiply two numbers, a and b, we're effectively multiplying all their factors together. For example, to do $42 \cdot 75 = 3150$, we can rewrite it as follows:

$42 = 2 \cdot 3 \cdot 7$
$75 = 3 \cdot 5 \cdot 5$
$42 \cdot 75 = 2 \cdot 3 \cdot 7 \cdot 3 \cdot 5 \cdot 5$

Now, in order to determine how many zeroes are on the end, we should look at how many complete pairs of 2 and 5 are among the factors. In the case of the example above, we have one 2 and two 5s, giving us **one** complete pair.

So, how does this relate to factorials? Well, in a factorial we're multiplying *all* the numbers between 1 and n together, which is the same as multiplying all the factors of the numbers between 1 and n.

For example, if $n = 16$, we need to look at the factors of all the numbers between 1 and 16. Keeping in mind that only 2s and 5s are of interest, we'll focus on those factors only. The numbers that contain a factor of 5 are $5, 10, 15$. The numbers that contain a factor of 2 are $2, 4, 6, 8, 10, 12, 14, 16$. Because there are only three numbers with a factor of 5, we can make three complete pairs, and therefore there must be three zeroes on the end of $16!$.

Putting this into an algorithm, we get:

```
twos = 0
for i from 1 to n inclusive:
    if i is divisible by 2:
        twos += 1

fives = 0
for i from 1 to n inclusive:
    if i is divisible by 5:
        fives += 1

tens = min(fives, twos)
```

This gets us most of the way, but it doesn't consider numbers with more than one factor. For example, if `i = 25`, then we've only done `fives += 1`. However, we should've done `fives += 2`, because 25 has *two* factors of 5.

Therefore, we need to count the 5 factors in each number. One way we can do this is by having a loop instead of the if statement, where each time we determine `i` has a 5 factor, we divide that 5 out. The loop will then repeat if there are further remaining 5 factors.

We can do that like this:

```
twos = 0
for i from 1 to n inclusive:
    remaining_i = i
    while remaining_i is divisible by 2:
        twos += 1
        remaining_i = remaining_i / 2

fives = 0
for i from 1 to n inclusive:
    remaining_i = i
    while remaining_i is divisible by 5:
        fives += 1
        remaining_i = remaining_i / 5

tens = min(twos, fives)
```

This gives us the right answer now. However, there are still some improvements we can make.

Firstly, we can notice that `twos` is **always bigger** than `fives`. Why? Well, every second number counts for a 2 factor, but only every fifth number counts as a 5 factor. Similarly every 4th number counts as an additional 2 factor, yet only every 25th number counts an additional 5 factor. This goes on and on for each power of 2 and 5. Here's a visualisation that illustrates how the density between 2 factors and 5 factors differs.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

As such, we can simply remove the whole `twos` calculation, leaving us with:

```
fives = 0
for i from 1 to n inclusive:
    remaining_i = i
    while remaining_i is divisible by 5:
        fives += 1
        remaining_i = remaining_i / 5

tens = fives
```

There is one final optimization we can do. In the above algorithm, we analyzed *every* number from 1 to n. However, only $5, 10, 15, 20, 25, 30, ...etc$ even have at least one factor of 5. So, instead of going up in steps of 1, we can go up in steps of 5. Making this modification gives us:

```
fives = 0
for i from 5 to n inclusive in steps of 5:
    remaining_i = i
    while remaining_i is divisible by 5:
        fives += 1
        remaining_i = remaining_i / 5

tens = fives
```

Algorithm

Here's the algorithm as we designed it above.

Java Python Copy

```python
def trailingZeroes(self, n: int) -> int:

    zero_count = 0
    for i in range(5, n + 1, 5):
        current = i
        while current % 5 == 0:
            zero_count += 1
            current //= 5

    return zero_count
```

Alternatively, instead of dividing by 5 each time, we can check each power of 5 to count how many times 5 is a factor. This works by checking if `i` is divisible by 5, then 25, then 125, etc. We stop when this number does not divide into `i` without leaving a remainder. The number of times we can do this is equivalent to the number of 5 factors in `i`.

Java Python Copy

```python
def trailingZeroes(self, n: int) -> int:

    zero_count = 0
    for i in range(5, n + 1, 5):
        power_of_5 = 5
        while i % power_of_5 == 0:
            zero_count += 1
            power_of_5 *= 5

    return zero_count
```

Complexity Analysis

- Time complexity : $O(n)$.

  In Approach 1, we couldn't treat division as $O(1)$, because we went well outside the 32-bit integer range. In this Approach though, we stay within it, and so can treat division and multiplication as $O(1)$.

  To calculate the zero count, we loop through every fifth number from 5 to n, this is $O(n)$ steps (the $\frac{1}{5}$ is treated as a constant).

  At each step, while it might look like we do a, $O(\log n)$ operation to count the number of fives, it actually amortizes to $O(1)$, because the vast majority of numbers checked only contain a single factor of 5. It can be proven that the total number of fives is less than $\frac{2 \cdot n}{5}$.

  So we get $O(n) \cdot O(1) = O(n)$.

- Space complexity : $O(1)$.

  We use only a fixed number of integer variables, therefore the space complexity is $O(1)$.

Approach 3: Counting Factors of 5 Efficiently

Intuition

In Approach 2, we found a way to count the number of zeroes in the factorial, *without* actually calculating the factorial. This was by looping over each multiple of 5, from 5 up to n, and counting how many factors of 5 were in each multiple of 5. We added all these counts together to get our final result.

However, Approach 2 was still too slow, both for practical means, and for the requirements of the question. To come up with a sufficient algorithm, we need to make one final observation. This observation will allow us to calculate our answer in logarithmic time.

Consider our simplified (but incorrect) algorithm that counted each multiple of 5. Recall that the reason it's incorrect is because it won't count both the 5 factors in numbers such as 25, for example.

```
fives = 0
for i from 1 to n inclusive:
    if i is divisible by 5:
        fives += 1
```

If you think about this overly simplified algorithm a little, you might notice that this is simply an inefficient way of performing integer division for $\frac{n}{5}$. Why? Well, by counting the number of multiples of 5 up to n, we're just counting how many 5s go into n. That's the *exact* definition of integer division!

So, a way of simplifying the above algorithm is as follows.

```
fives = n / 5
tens = fives
```

So, how can we fix the "duplicate factors" problem? Observe that *all* numbers that have (at least) two factors of 5 are multiples of 25. Like with the 5 factors, we can simply divide by 25 to find how many multiples of 25 are below n. Also, notice that because we've already counted the multiples of 25 in $\frac{n}{5}$ once, we only need to count $\frac{n}{25}$ extra factors of 5 (not $2 \cdot \frac{n}{25}$), as this is one extra for each multiple of 25.

So combining this together we get:

```
fives = n / 5 + n / 25
tens = fives
```

We still aren't there yet though! What about the numbers which contain *three* factors of 5 (the multiples of 125). We've only counted them twice! In order to get our final result, we'll need to add together all of $\frac{n}{5}$, $\frac{n}{25}$, $\frac{n}{125}$, $\frac{n}{625}$, and so on. This gives us:

$$fives = \frac{n}{5} + \frac{n}{25} + \frac{n}{125} + \frac{n}{625} + \frac{n}{3125} + \cdots$$

This might look like it goes on forever, but it doesn't! Remember that we're using **integer division**. Eventually, the denominator will be *larger than* n, and so all the terms from there will be 0. Therefore, we can stop once the term is 0.

For example with $n = 12345$ we get:

$$fives = \frac{12345}{5} + \frac{12345}{25} + \frac{12345}{125} + \frac{12345}{625} + \frac{12345}{3125} + \frac{12345}{16075} + \frac{12345}{80375} + \cdots$$

Which is equal to:

$$fives = 2469 + 493 + 98 + 19 + 3 + 0 + 0 + \cdots = 3082$$

In code, we can do this by looping over each power of 5, calculating how many times it divides into n, and then adding that to a running `fives` count. Once we have a power of 5 that's bigger than n, we stop and return the final value of `fives`.

```
fives = 0
power_of_5 = 5
while n >= power_of_5:
    fives += n / power_of_5
    power_of_5 *= 5

tens = fives
```

Algorithm

Java Python Copy

```python
def trailingZeroes(self, n: int) -> int:
    zero_count = 0
    current_multiple = 5
    while n >= current_multiple:
        zero_count += n // current_multiple
        current_multiple *= 5
    return zero_count
```

An alternative way of writing this algorithm, is instead of trying each power of 5, we can instead divide n itself by 5 each time. This works out the same because we wind up with the sequence:

$$fives = \frac{n}{5} + \frac{(\frac{n}{5})}{5} + \frac{(\frac{(\frac{n}{5})}{5})}{5} + \cdots$$

Notice that on the second step, we have $\frac{(\frac{n}{5})}{5}$. This is because the previous step divided n itself by 5. And so on.

If you're familiar with the rules of fractions, you'll notice that $\frac{(\frac{n}{5})}{5}$ is just the same thing as $\frac{n}{5 \cdot 5} = \frac{n}{25}$. This means the sequence is exactly the same as:

$$\frac{n}{5} + \frac{n}{25} + \frac{n}{125} + \cdots$$

So, this alternative way of writing the algorithm is equivalent.

Java Python Copy

```python
def trailingZeroes(self, n: int) -> int:
    zero_count = 0
    while n > 0:
        n //= 5
        zero_count += n
    return zero_count
```

Complexity Analysis

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

  In this approach, we divide n by each power of 5. By definition, there are $\log_5 n$ powers of 5 less-than-or-equal-to n. Because the multiplications and divisions are within the 32-bit integer range, we treat these calculations as $O(1)$. Therefore, we are doing $\log_5 n \cdot O(1) = \log n$ operations (keeping in mind that $\log$ bases are insignificant in big-oh notation).

- Space complexity : $O(1)$.

  We use only a fixed number of integer variables, therefore the space complexity is $O(1)$.

Previous Next