Previous Next

264. Ugly Number II

July 13, 2019 | 20.4K views

Average Rating: 2.82 (49 votes)

Write a program to find the `n` -th ugly number.

Ugly numbers are **positive numbers** whose prime factors only include `2, 3, 5`.

Example:

```
Input: n = 10
Output: 12
Explanation: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10
 ugly numbers.
```

Note:

1. `1` is typically treated as an ugly number.
2. `n` **does not exceed 1690**.

Solution

Two levels of optimisation

Let's imagine that the problem is solved somehow for the number n and we've put the solution directly in `nthUglyNumber` method of the `Solution` class.

Now let's check the context: there are 596 test cases, for the most of them n is larger than 50, and n is known to be smaller than 1691.

Hence instead of computing $596 \times 50 = 29800$ ugly numbers in total, one could precompute all 1690 numbers, and significantly speed up the submission.

> How to precompute? Use another class `Ugly` with all computations in the constructor and then declare `Ugly` instance as a static variable of `Solution` class.

Now let's consider two different approaches to perform the preliminary computations.

Approach 1: Heap

Intuition

Let's start from the heap which contains just one number: 1.
To compute next ugly numbers, pop 1 from the heap and push instead three numbers: 1×2, 1×3, and 1×5.

Now the smallest number in the heap is 2. To compute next ugly numbers, pop 2 from the heap and push instead three numbers: 2×2, 2×3, and 2×5.

> One could continue like this to compute first 1690 ugly numbers. At each step, pop the smallest ugly number k from the heap, and push instead three ugly numbers: $k \times 2$, $k \times 3$, and $k \times 5$.

Algorithm

- Precompute 1690 ugly numbers:
  - Initiate array of precomputed ugly numbers `nums`, heap `heap` and hashset `seen` to track all elements already pushed in the heap in order to avoid duplicates.
  - Make a loop of 1690 steps. At each step:
    - Pop the smallest element `k` out of heap and add it into the array of precomputed ugly numbers.
    - Push `2k`, `3k` and `5k` in the heap if they are not yet in the hashset. Update the hashset of seen ugly numbers as well.
- Retrieve needed ugly number from the array of precomputed numbers.

Implementation

Java Python Copy

```python
from heapq import heappop, heappush
class Ugly:
    def __init__(self):
        seen = {1, }
        self.nums = nums = []
        heap = []
        heappush(heap, 1)

        for _ in range(1690):
            curr_ugly = heappop(heap)
            nums.append(curr_ugly)
            for i in [2, 3, 5]:
                new_ugly = curr_ugly * i
                if new_ugly not in seen:
                    seen.add(new_ugly)
                    heappush(heap, new_ugly)

class Solution:
    u = Ugly()
    def nthUglyNumber(self, n):
        return self.u.nums[n - 1]
```

Complexity Analysis

- Time complexity : $\mathcal{O}(1)$ to retrieve preliminary computed ugly number, and more than 12×10^6 operations for preliminary computations. Let's estimate the number of operations needed for the preliminary computations. `For` loop here has 1690 steps, and each step performs 1 pop, not more than 3 pushes and 3 `contains / in` operations for the hashset. Pop and push have logarithmic time complexity and hence much cheaper than the linear search, so let's estimate only the last term. This arithmetic progression is easy to estimate:

$$1 + 2 + 3 + ... + 1690 \times 3 = \frac{(1 + 1690 \times 3) \times 1690 \times 3}{2} > 4.5 \times 1690^2$$

- Space complexity : constant space to keep an array of 1690 ugly numbers, the heap of not more than 1690×2 elements and the hashset of not more than 1690×3 elements.

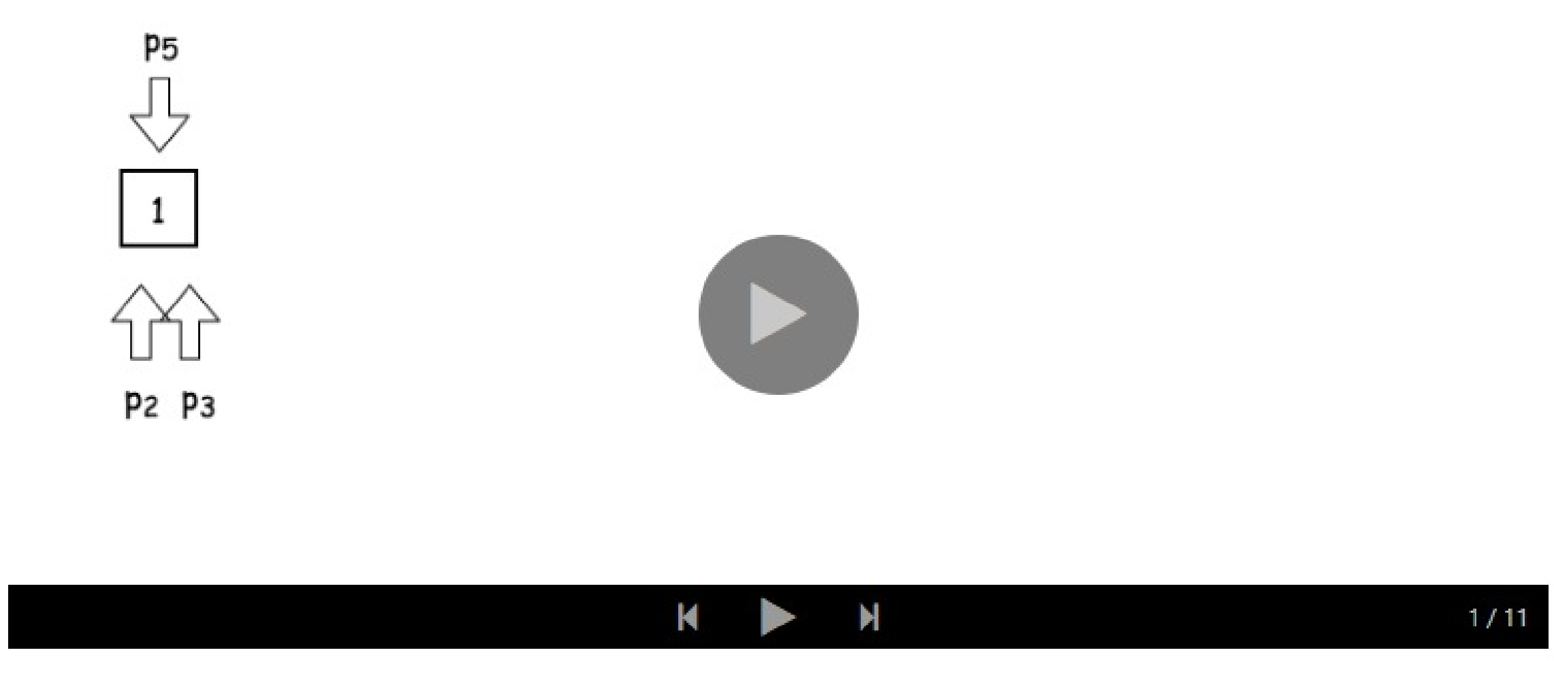
Approach 2: Dynamic Programming

Intuition

Preliminary computations in Approach 1 are quite heavy, and could be optimised with dynamic programming.

Let's start from the array of ugly numbers which contains just one number - 1. Let's use three pointers i_2, i_3 and i_5, to mark the last ugly number which was multiplied by 2, 3 and 5, correspondingly.

The algorithm is straightforward: choose the smallest ugly number among $2 \times \textrm{nums}[i_2]$, $3 \times \textrm{nums}[i_3]$, and $5 \times \textrm{nums}[i_5]$ and add it into the array. Move the corresponding pointer by one step. Repeat till you'll have 1690 ugly numbers.



Algorithm

- Precompute 1690 ugly numbers:
  - Initiate array of precomputed ugly numbers `nums` and three pointers `i2`, `i3` and `i5` to track the index of the last ugly number used to produce the next ones.
  - Make a loop of 1690 steps. At each step:
    - Choose the smallest number among `nums[i2] * 2`, `nums[i3] * 3`, and `nums[i5] * 5` and add it into `nums`.
    - Move by one the pointer which corresponds to the "ancestor" of the added number.
- Retrieve needed ugly number from the array of precomputed numbers.

Implementation

Java Python Copy

```python
class Ugly:
    def __init__(self):
        self.nums = nums = [1, ]
        i2 = i3 = i5 = 0

        for i in range(1, 1690):
            ugly = min(nums[i2] * 2, nums[i3] * 3, nums[i5] * 5)
            nums.append(ugly)

            if ugly == nums[i2] * 2:
                i2 += 1
            if ugly == nums[i3] * 3:
                i3 += 1
            if ugly == nums[i5] * 5:
                i5 += 1

class Solution:
    u = Ugly()
    def nthUglyNumber(self, n):
        return self.u.nums[n - 1]
```

Complexity Analysis

- Time complexity : $\mathcal{O}(1)$ to retrieve preliminary computed ugly number, and about $1690 \times 5 = 8450$ operations for preliminary computations.
- Space complexity : constant space to keep an array of 1690 ugly numbers.

Rate this article: ★★★☆☆

Previous Next

Comments: 20 Sort By

Type comment here... (Markdown is supported)

Preview Post

edzvh ★ 130 March 7, 2020 6:12 PM

Whoever wrote the analysis for solution 1 seems to think that set `contains` is `O(n)` complexity. Seriously guys, people are paying for this content. Does no one even proof read the articles?

30 Share Reply

SHOW 5 REPLIES

pmane4422 ★ 255 July 5, 2020 2:15 AM

Don't you think it is unfair to make constructors that heavy? Also precomputing all results under given constraints and calling it O(1) is a cheating

9 Share Reply

xi31 ★ 32 December 6, 2019 9:46 AM

I think the second approach is more like three-pointer.

9 Share Reply

agacooker ★ 11 August 18, 2019 8:18 AM

Where is the protection to prevent nums to have repeat numbers?

7 Share Reply

SHOW 5 REPLIES

nickyflash ★ 20 July 4, 2020 11:02 PM

Precompute is almost cheating, it won't work when you don't know the upper limit of N. In that generalized case, the heap approach will have a time complexity of O(n*log n) and the DP approach will have a time complexity O(n).

3 Share Reply

SHOW 1 REPLY

fbma ★ 39 July 4, 2020 1:02 PM

The time complexity for the first approach is terribly wrong as said by the other users here, but other than that, why would you pre-compute the 1690 first elements if you just need the n-th? This would be good if you would have a few amount of queries, but since you have only one query per call it makes no sense, and it would be better optimized if you get only the n-th first elements.

Read More

3 Share Reply

quantumlexa ★ 27 July 6, 2020 6:45 AM

explanation of DP approach as ugly as all of those numbers together

2 Share Reply

ssl_91 ★ 28 October 16, 2019 1:57 PM

@andvary Nice solution. How did you approach solution (2) and what's the intuition on using these pointers? I couldn't understand until I traced it on paper :)

2 Share Reply

SHOW 1 REPLY

mangrer ★ 219 July 5, 2020 9:28 PM

You'll look bad if you attempt to precompute all 1690 elements in an interview setting, compute up to N using a heap and mention the time complexity as O (N), and a heap would cost log (NK).

1 Share Reply

zhang-peter ★ 26 September 22, 2019 9:27 PM

my python solution:

```
class Solution:
    def nthUglyNumber(self, n: int) -> int:
        if n == 1:
```

Read More

0 Share Reply

SHOW 2 REPLIES

‹ 1 2 ›