

Algorithms

December 28, 2018

Contents

1	Linked List	2
2	Sliding Window Technique	3
2.1	Count distinct elements in every window of size k	3
2.2	Sliding Window Maximum (Maximum of all subarrays of size k)	4
3	Heap	5
3.1	Python's heapq	7
3.2	The application of heap	10
3.2.1	Merge k Sorted Lists (LeetCode 23)	10
3.2.2	Kth Largest Element in a Stream (LeetCode 703) . . .	13
3.2.3	Top K Frequent Elements (LeetCode 347)	14
4	Sorting	17
4.1	Sort Characters By Frequency (LeetCode 451)	17
5	Dynamic Programming	19
6	Maths	19
6.1	Prime numbers	19
6.1.1	Count Primes (LeetCode 204)	19
6.1.2	Ugly Number (LeetCode 263)	21
6.1.3	Ugly Number II (LeetCode 264)	23
6.1.4	Super Ugly Number (LeetCode 313)	26
6.2	Permutation	27
6.2.1	Next Permutation (LeetCode 31)	27

1 Linked List

We can use array to initialize the linked list as the following code. While building up the linked list from the given array, we use the **two-pointer technique** to maintain the linking between two nodes.

```
1  #Definition for singly-linked list.
2  class ListNode(object):
3      def __init__(self, x):
4          self.val = x
5          self.next = None
6
7
8  class List(object):
9      def __init__(self, array):
10         if(array):
11             self.head = ListNode(array[0])
12             prev=self.head
13             cur = None
14             for i in range(1,len(array)):
15                 cur=ListNode(array[i])
16                 prev.next=cur
17                 prev=cur
18         else:
19             self.head = None
20
21     def __str__(self):
22         if(not self.head):
23             return "[]"
24         else:
25             s="["
26             cur=self.head
27             while(cur):
28                 s=s+str(cur.val)+" "
29                 cur=cur.next
30             s=s.strip()+"]"
31             return s
32
33
34 array1=[1,4,5]
```

```

35 list1=List(array1)
36 print(list1)

```

2 Sliding Window Technique

2.1 Count distinct elements in every window of size k

Tag: Sliding Window Technique, Hashtable. See ¹.

Input: `arr[] = {1, 2, 1, 3, 4, 2, 3}`, `k = 4`
 Output: `[3, 4, 4, 3]`

We use the sliding window to update a hashtable, which maintains the distinct elements. And the time complexity is $O(n)$.

```

class Solution():
    '''2018-12-21
    '''
    def distinct(self, nums, k):
        if(not nums or len(nums)<k):
            return []
        d=dict()
        res=[]
        #init the first window
        for j in range(0,k):
            if(nums[j] not in d):
                d[nums[j]]=1
            else:
                d[nums[j]]+=1
        res.append(len(d))

        #update the remaining windows
        for i in range(1,len(nums)-k+1):
            #remove the first in the window, and add the last
            #to the window.
            first=nums[i-1]
            last=nums[i+k-1]
            if(d[first]==1):

```

¹<https://www.geeksforgeeks.org/count-distinct-elements-in-every-window-of-size-k/>

```

        d.pop(first)
    else:
        d[first]-=1

    if(last not in d):
        d[last]=1
    else:
        d[last]+=1

    res.append(len(d))

return res

def testAll(self):
    testcase1={"nums":[1, 2, 1, 3, 4, 2, 3],"k":4,"expected":[3,4,4,3]}
    testcase2={"nums":[1, 2, 1],"k":4,"expected":[]}
    testcase3={"nums":[1, 2, 1, 3, 4, 2, 3, 5],"k":4,"expected":[3,4,4,3,4]}
    testcases=[testcase1,testcase2,testcase3]
    for testcase in testcases:
        self.test(testcase["nums"],testcase["k"],testcase["expected"])

def test(self,nums,k,expected):
    res=self.distinct(nums,k)
    print("Test on nums={0}, k={1}. And {2} is expected, and {3} is got."\
        .format(nums,k,expected,res))

```

```

a=Solution()
a.testAll()

```

2.2 Sliding Window Maximum (Maximum of all subarrays of size k)

See ².

²<https://www.geeksforgeeks.org/sliding-window-maximum-maximum-of-all-subarrays-of-size-k/>

```

Input :
arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}
k = 3
Output :
3 3 4 5 5 5 6

Input :
arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}
k = 4
Output :
10 10 10 15 15 90 90

```

We use the priority queue to .

```

class Solution():
    """2018-12-21
    """
    def maxSlidingWindow(self, nums, k):
        pass

```

3 Heap

Heap can be viewed as a complete tree, but stored as the array. Suppose the current node's index is idx , then the left child's index is $2 * idx + 1$, and the right child $2 * idx + 2$, while the parent $\text{floor}((idx - 1)/2)$.

We take the binary max heap as an example. The basic external function is **insert** and **extractMax**, which is implemented by **siftup** and **siftdown**. The **siftup** function check the current node's value with its parent's value, then swap them if the current node's value is bigger than the parent's, and do the check-swap operation recursively to meet the guarantee of the binary max heap.

The python source code is as following.

```

1 class MaxHeap():
2     '''
3     2018-12-24
4     The root is bigger than its left child and right child.
5     '''
6     def __init__(self):
7         self.array=[]

```

```

8
9     def insert(self,num):
10         if(not self.array):
11             self.array.append(num)
12         else:
13             self.array.append(num)
14             self.siftup(len(self.array)-1)
15
16
17     def siftup(self,idx):
18         if(idx==0):
19             return
20         parentIdx=(idx - 1) // 2
21         if(self.array[parentIdx]<self.array[idx]):
22             self.array[parentIdx], self.array[idx] = \
23                 self.array[idx], self.array[parentIdx]
24             return self.siftup(parentIdx)
25
26     def extractMax(self):
27         #swap the head (max) and the last one in self.array,
28         ↪ then pop out the max
29         if(not self.array):
30             raise ValueError("pop out from an empty heap")
31
32         ↪ self.array[0],self.array[-1]=self.array[-1],self.array[0]
33         max=self.array.pop()
34         self.siftdown(0)
35         return max
36
37     def siftdown(self,idx):
38         '''
39         move the current node down
40         '''
41         if(not self.array or len(self.array)==1):
42             return
43         left=idx*2+1
44         right=idx*2+2
45         maxIdx=idx

```

```

46         if(left<len(self.array) and
           ↪ self.array[maxIdx]<self.array[left]):
47             maxIdx=left
48         if(right<len(self.array) and
           ↪ self.array[maxIdx]<self.array[right]):
49             maxIdx=right
50
51         self.array[idx], self.array[maxIdx] =
           ↪ self.array[maxIdx], self.array[idx]
52         #sift down the smaller number recursively
53         if(idx!=maxIdx):
54             self.siftdown(maxIdx)
55
56
57     def __str__(self):
58         s=""
59         for i in range(len(self.array)):
60             if(i!=len(self.array)-1):
61                 s+=str(self.array[i])+" "
62             else:
63                 s+=str(self.array[i])
64         return s
65
66
67 heap=MaxHeap()
68 for i in range(1,10):
69     heap.insert(i)
70 print("After inserting 1,2,3,4,5,6,7,8,9, the array of the
     ↪ heap is {0}.".format(heap))
71
72 maxInHeap=heap.extractMax()
73 print("Pop out from the heap, we'll get the maximum number
     ↪ {0}, "
74       "and the array of the heap becomes
     ↪ {1}.".format(maxInHeap,heap))

```

3.1 Python's heapq

We can use the library **heapq** in python. Since the default **heapq** is the min heap, so we need a trick to reimplement **MaxHeap** by overriding the

comparison function.

```
1 import heapq
2 '''2018-12-24
3 Use python's heapq to implement a binary max heap.
4 '''
5
6 class MaxHeapObj(object):
7     def __init__(self, val):
8         self.val = val
9
10    def __lt__(self, other):
11        return self.val > other.val
12
13    def __eq__(self, other):
14        return self.val == other.val
15
16    def __str__(self):
17        return str(self.val)
18
19
20 class MaxHeap(object):
21     def __init__(self):
22         self.h = []
23
24     def heappush(self, x):
25         heapq.heappush(self.h, MaxHeapObj(x))
26
27     def heappop(self):
28         return heapq.heappop(self.h).val
29
30     def __getitem__(self, i):
31         return self.h[i].val
32
33     def __str__(self):
34         s=""
35         for e in self.h:
36             s=s+str(e)+" "
37         return s
38
```



```

39
40 heap=MaxHeap()
41
42 for i in range(1,10):
43     heap.heappush(i)
44
45 print("After inserting 1,2,3,4,5,6,7,8,9, the array of the
↳ heap is {0}.".format(heap))
46
47 maxInHeap=heap.heappop()
48 print("Pop out from the heap, we'll get the maximum number
↳ {0}, "
49     "and the array of the heap becomes
↳ {1}.".format(maxInHeap,heap))

```

Or we can implement **MaxHeap** by multiplying -1 to each item in an array directly when using **heapq**.

```

1 import heapq
2 '''2018-12-24
3 Use python's heapq to implement a binary max heap.
4 Since heapq is the min heap, so we need to reimplement
5 MaxHeap by multiplying -1 to the item in min heap, and
6 multiply -1 as well when using heappop() funciton.
7 '''
8
9 class MaxHeap(object):
10     def __init__(self):
11         self.h = []
12
13     def heappush(self,x):
14         heapq.heappush(self.h,-x)
15
16     def heappop(self):
17         return heapq.heappop(self.h)*(-1)
18
19     def __getitem__(self, i):
20         return -1*self.h[i]
21
22     def __str__(self):
23         s=""

```

```

24         for i in range(len(self.h)):
25             s=s+str(self[i])+" "
26         return s
27
28
29 heap=MaxHeap()
30
31 for i in range(1,10):
32     heap.heappush(i)
33
34 print("After inserting 1,2,3,4,5,6,7,8,9, the array of the
    ↪ heap is {0}.".format(heap))
35
36 maxInHeap=heap.heappop()
37 print("Pop out from the heap, we'll get the maximum number
    ↪ {0}, "
38       "and the array of the heap becomes
    ↪ {1}.".format(maxInHeap,heap))

```

All the above three heap codes generate the following output.

```

After inserting 1,2,3,4,5,6,7,8,9, the array of the heap
↪ is 9 8 6 7 3 2 5 1 4 .
Pop out from the heap, we'll get the maximum number 9,
↪ and the array of the heap becomes 8 7 6 4 3 2 5 1 .

```

3.2 The application of heap

3.2.1 Merge k Sorted Lists (LeetCode 23)

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity³.

³<https://leetcode.com/problems/merge-k-sorted-lists/description/>

Input:

```
[
    1->4->5,
    1->3->4,
    2->6
]
```

Output: 1->1->2->3->4->4->5->6

```
1 import heapq
2
3 # Definition for singly-linked list.
4 class ListNode(object):
5     def __init__(self, x):
6         self.val = x
7         self.next = None
8
9 class List(object):
10     def __init__(self, array):
11         if array:
12             self.head = ListNode(array[0])
13             prev = self.head
14             cur = None
15             for i in range(1, len(array)):
16                 cur = ListNode(array[i])
17                 prev.next = cur
18                 prev = cur
19         else:
20             self.head = None
21
22 class Solution(object):
23     def mergeKLists(self, lists):
24         """
25         :type lists: List[ListNode]
26         :rtype: ListNode
27         """
28         heap = []
29         #init heap, which contains the tuple of head.val and
30         ↪ head of each list
31         for list in lists:
```

```

31         if(list):
32             heapq.heappush(heap, (list.val, list))
33
34         dummy=ListNode(0)
35         tail_new_list=dummy
36         #update
37         while(heap):
38             _,head=heapq.heappop(heap)
39             if(head.next):
40                 heapq.heappush(heap, (head.next.val, head.next))
41
42             tail_new_list.next=head
43             tail_new_list=tail_new_list.next
44         return dummy.next
45
46 def display(head):
47     s=""
48     cur=head
49     while(cur):
50         s+=str(cur.val)+" "
51         cur=cur.next
52     return s
53
54
55 array1=[1,4,5]
56 list1=List(array1)
57 print(list1)
58
59 array2=[1,3,4]
60 list2=List(array2)
61 print(list2)
62
63 array3=[2,6]
64 list3=List(array3)
65 print(list3)
66
67
68 solution=Solution()
69 mergedList=solution.mergeKLists([list1.head, list2.head, list3.head])
70 print(display(mergedList))

```

3.2.2 Kth Largest Element in a Stream (LeetCode 703)

Design a class to find the kth largest element in a stream. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Your KthLargest class will have a constructor which accepts an integer k and an integer array nums, which contains initial elements from the stream. For each call to the method KthLargest.add, return the element representing the kth largest element in the stream.

Example:

```
int k = 3;
int[] arr = [4,5,8,2];
KthLargest kthLargest = new KthLargest(3, arr);
kthLargest.add(3);    // returns 4
kthLargest.add(5);    // returns 5
kthLargest.add(10);   // returns 5
kthLargest.add(9);    // returns 8
kthLargest.add(4);    // returns 8
```

Note:

1. You may assume that nums' length $\geq k - 1$ and $k \geq 1$.

We use the min heap with the fixed size k to maintain the largest k elements in the stream. The minimum element in the min heap with size k will be the k-th largest element in a stream. The initialization of the heap is to heapify the given array. Since the time complexity of the operation of "heapify" is $O(n)$ ⁴, so the initialization is very efficient. And the add operation in the following code is to pop out the minimum element in the array which costs $O(\log k)$. In general, the time complexity is $O(n \log k)$, where n is the size of the stream, and the space complexity is $O(\log k)$.

```
1 import heapq
2
3 class KthLargest(object):
4     '''
5     2018-12-26
6     '''
7     def __init__(self, k, nums):
```

⁴<https://www.growingwiththeweb.com/data-structures/binary-heap/build-heap-proof/>

```

8         """
9         :type k: int
10        :type nums: List[int]
11        """
12        #self.array is the min heap
13        self.heap=nums
14        heapq.heapify(self.heap)
15        self.k=k
16        while(len(self.heap)>k):
17            heapq.heappop(self.heap)
18
19
20    def add(self, val):
21        """
22        :type val: int
23        :rtype: int
24        """
25        if(len(self.heap)<self.k):
26            heapq.heappush(self.heap, val)
27        else:
28            if(val>self.heap[0]):
29                #update min heap
30                heapq.heappop(self.heap)
31                heapq.heappush(self.heap, val)
32        return self.heap[0]
33
34    # Your KthLargest object will be instantiated and called as
35    ↪ such:
36    # obj = KthLargest(k, nums)
37    # param_1 = obj.add(val)

```

3.2.3 Top K Frequent Elements (LeetCode 347)

Given a non-empty array of integers, return the k most frequent elements.

Example 1:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`
Output: `[1,2]`

Example 2:

Input: `nums = [1], k = 1`
Output: `[1]`

Note:

1. You may assume k is always valid, $1 \leq k \leq$ number of unique elements.
2. Your algorithm's time complexity must be better than $O(n \log n)$, where n is the array's size.

There are two solutions for this problem: **min heap** and **max heap**. As the first solution, we use the **min heap** with the fixed size k to maintain the top k frequent elements, which is inspired by the problem **Kth Largest Element in an Array**. Taking the counting frequent, heapifying the frequent-num array, and popping out the results into consideration, the time complexity is $O(n) + O(k + (n - k) \log k) + O(k \log k)$. So if $n \gg k$, the overall time complexity is $O(n \log k)$, but if $n \approx k$, the overall time complexity is $O(n \log n)$.

```
1 import heapq
2 import collections
3
4 class Solution(object):
5     '''
6     2018-12-27
7     min heap method
8     '''
9     def topKFrequent(self, nums, k):
10        """
11        :type nums: List[int]
12        :type k: int
13        :rtype: List[int]
14        """
15        #firstly, count the frequent.
16        #The time complexity is O(n).
17        numfreq=collections.defaultdict(int)
18        for num in nums:
19            numfreq[num]+=1
20
21        #secondly, use the min heap with
22        #the fixed size k to get the top k frequent
```

```

23         #numbers. The time complexity is O(n * log k)
24         heap=[]
25         for num in numfreq:
26             freq=numfreq[num]
27             if(len(heap)<k):
28                 heapq.heappush(heap, (freq,num))
29             else:
30                 if(freq>heap[0][0]):
31                     heapq.heappop(heap)
32                     heapq.heappush(heap, (freq,num))
33
34         #thirdly, output the result from the min heap.
35         #The time complexity is O(k * log k).
36         res=[]
37         while(heap):
38             num=heapq.heappop(heap)[1]
39             res.append(num)
40         res.reverse()
41         return res

```

As the second solution, we use the **max heap** to heapify the frequent-num tuples to a heap with the size n . Then we pop out the first k elements in the max heap, which will be the result. Similar as the analysis on the min heap method, taking the counting frequent, heapifying the frequent-num array, and popping out the results into consideration, the time complexity is $O(n) + O(n) + O(k \log n)$. So if $n \gg k$, the overall time complexity is $O(n + k \log n)$ ($\approx O(n)$), but if $n \approx k$, the overall time complexity is $O(n \log n)$.

```

1  import heapq
2  import collections
3
4  class Solution(object):
5      '''
6      2018-12-27
7      max heap method
8      '''
9      def topKFrequent(self, nums, k):
10         """
11         :type nums: List[int]

```



```

12         :type k: int
13         :rtype: List[int]
14         """
15         #firstly, count the frequent.
16         #The time complexity is O(n).
17         numfreq=collections.defaultdict(int)
18         for num in nums:
19             numfreq[num]+=1
20
21         #secondly, use the max heap with
22         #the size n to heapify the frequent-number tuples
23         #The time complexity is O(n)
24         heap=[(-1*freq,num) for num,freq in numfreq.items()]
25         heapq.heapify(heap)
26
27         #thirdly, pop out the top k frequent elements.
28         #The time complexity is O(k*log n)
29         res=[]
30         while(len(res)<k):
31             num=heapq.heappop(heap)[1]
32             res.append(num)
33         return res

```

4 Sorting

4.1 Sort Characters By Frequency (LeetCode 451)

Given a string, sort it in decreasing order based on the frequency of characters.

Example 1:

Input:

"tree"

Output:

"eert"

Explanation:

'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore
↪ "eetr" is also a valid answer.

Example 2:

Input:

"cccaa"

Output:

"cccaa"

Explanation:

Both 'c' and 'a' appear three times, so "aaacc" is
↪ also a valid answer.

Note that "cacaca" is incorrect, as the same
↪ characters must be together.

Example 3:

Input:

"Aabb"

Output:

"bbAa"

Explanation:

"bbaA" is also a valid answer, but "Aabb" is
↪ incorrect.

Note that 'A' and 'a' are treated as two different
↪ characters.

This problem is not difficult. But we still should pay attention to the potential TLE problem may caused by the combination of string in python. We cannot use "s=s+c" which time complexity is $O(n^2)$ and space complexity is also $O(n^2)$ since it needs deep copy for every adding operation.

```
1 import collections
2
3 class Solution(object):
4     '''
5     2018-12-28
6     '''
7     def frequencySort(self, s):
8         """
9         :type s: str
10        :rtype: str
11        """
12        # count
13        charFreq = collections.defaultdict(int)
14        for c in s:
15            charFreq[c] += 1
16
17        # sort
18        tuples = [(freq, c) for c, freq in charFreq.items()]
19        tuples.sort(reverse=True)
20
21        res = []
22        for freq, c in tuples:
23            res += [c]*freq
24
25        return "".join(res)
```

5 Dynamic Programming

6 Maths

6.1 Prime numbers

6.1.1 Count Primes (LeetCode 204)

Count the number of prime numbers **less than** a non-negative number, n.
Example:

Input: 10
Output: 4
Explanation: There are 4 prime numbers less than 10,
↪ they are 2, 3, 5, 7.

Tag: Primes.

We use the **Sieve of Eratosthenes** to label each number within the array of $[1, \dots, n]$ is a prime or not.

```
1 import math
2
3 class Solution(object):
4     '''2018-12-15
5     This is an O(n log n) solution.
6     '''
7     def countPrimes(self, n):
8         """
9         :type n: int
10        :rtype: int
11        """
12        if(n<=1):
13            return 0
14        #primeFlags[i]=True means the number i is a prime
15        ↪ number.
16        primeFlags=[True]*(n+1)
17        primeFlags[0]=False
18        primeFlags[1]=False
19
20        for p in range(2,int(math.sqrt(n))+1):
21            if(primeFlags[p]==True):
22                for multiplier in range(p,n//p+1):
23                    primeFlags[p*multiplier]=False
24        return sum(primeFlags[:-1])
25
26 def test(self):
27     print("n={0}, output={1},
28         ↪ expected={2}".format(5,self.countPrimes(5),2))
29     print("n={0}, output={1},
30         ↪ expected={2}".format(10,self.countPrimes(10),4))
```

```

28 |
29 | a=Solution()
30 | a.test()

```

The line 18 to line 19 in the code can be optimized as the following code without the time consumption on the loop.

```

primeFlags[p*p:n:p] =
↪ [False]*len(primeFlags[p*p:n:p])

```

By eliminating the inner loop, the time consumption is reduced from 860ms to 304ms.

6.1.2 Ugly Number (LeetCode 263)

Write a program to check whether a given number is an ugly number.

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

Example 1:

```

Input: 6
Output: true
Explanation: 6 = 2 * 3

```

Example 2:

```

Input: 8
Output: true
Explanation: 8 = 2 * 2 * 2

```

Example 3:

```

Input: 14
Output: false
Explanation: 14 is not ugly since it includes another
↪ prime factor 7.

```

Note:

1. 1 is typically treated as an ugly number.

2. Input is within the 32-bit signed integer range: $[-2^{31}, 2^{31} - 1]$.

We use the while loop to do the check and the decomposition for a given number. Since the given number is within the range $[-2^{31}, 2^{31} - 1]$, so we can do the check and the decomposition by recursion without worrying about the stack overflow (exceeding the maximum recursion depth).

```
1 class Solution(object):
2     '''2018-12-25
3     '''
4     def isUgly(self, num):
5         """
6         :type num: int
7         :rtype: bool
8         """
9         if (num < 1):
10             return False
11
12         if (num == 1):
13             return True
14
15         while (num > 1):
16             ugly = False
17             for p in [2, 3, 5]:
18                 if (num % p == 0):
19                     num = num / p
20                     ugly = True
21                     break
22             if (ugly == False):
23                 return False
24         return True
25
26
27 a=Solution()
28 print(a.isUgly(2147483648))
```

The recursion version is as following.

```
1 class Solution(object):
2     '''2018-12-25
3     '''
```

```

4      def isUgly(self, num):
5          """
6          :type num: int
7          :rtype: bool
8          """
9          def helper(num):
10             if(num<=0):
11                 return False
12             if(num==1):
13                 return True
14
15             if(num%2==0):
16                 return helper(num//2)
17             if(num%3==0):
18                 return helper(num//3)
19             if(num%5==0):
20                 return helper(num//5)
21             return False
22
23         return helper(num)

```

6.1.3 Ugly Number II (LeetCode 264)

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.

Tag: Maths, Primes, Tricky.

This problem's solution is very very tricky. Although the tag on this problem includes dynamic programming. But I don't think it's a good example of the dynamic programming technique, because it can not get a clear

recursive formula. Instead, I would rather call it a tricky solution only using a tabulation.

Suppose the resulted ugly number list is F . Since F includes the numbers whose factors only include 2, 3, and 5. So we build up three lists $l_2 = 2 * F$, $l_3 = 3 * F$, and $l_5 = 5 * F$. Therefore the ugly list F has the property that $F = [1] + \text{merge}(l_2, l_3, l_5)$. Based on this property, what we should do is merging l_2 , l_3 , and l_5 . The **tricky part** is that F should be merged from l_2 , l_3 , and l_5 , while these three lists are also need to be built from F . We handle it by updating them simultaneously as the following code.

```

1  class Solution(object):
2      '''
3      2018-12-25
4      '''
5      def nthUglyNumber(self, n):
6          """
7              :type n: int
8              :rtype: int
9              """
10         F=[0]*(n+1) #F is the list of ugly numbers
11         F[0]=0
12         F[1]=1
13         l2=[1*2] #l2 is the list of 2*F
14         l3=[1*3] #l3 is the list of 3*F
15         l5=[1*5] #l5 is the list of 5*F
16
17         cur2=0
18         cur3=0
19         cur5=0
20         for i in range(2,n+1):
21             #update F, since F is the merge of l2, l3, and l5
22             ↪ except for 0, and 1.
23             #So we apply the merging method to update F.
24             #And l2, l3, and l5 is based on F, so the update
25             ↪ is simultaneously,
26             #very tricky.
27             F[i]=min(l2[cur2],l3[cur3],l5[cur5])
28
29             #update l2, l3, and l5
30             l2.append(F[i] * 2)

```



```

29         l3.append(F[i] * 3)
30         l5.append(F[i] * 5)
31
32         #update the pointers to l2, l3, and l5
33         if(F[i]==l2[cur2]):
34             cur2+=1
35         if(F[i]==l3[cur3]):
36             cur3+=1
37         if(F[i]==l5[cur5]):
38             cur5+=1
39     return F[n]

```

If we want to save the space of l_2 , l_3 , and l_5 , then we'll do not claim space for them but use the space of F only by maintaining the pointers in these three lists. The code is as following.

```

1  class Solution(object):
2      '''
3      2018-12-25
4      '''
5      def nthUglyNumber(self, n):
6          """
7          :type n: int
8          :rtype: int
9          """
10         F=[0]*n
11         F[0]=1
12         n2=2 #the value of current node in l2
13         n3=3 #the value of current node in l3
14         n5=5 #the value of current node in l5
15
16         cur2=0 #the pointer to the current node in l2
17         cur3=0 #the pointer to the current node in l3
18         cur5=0 #the pointer to the current node in l5
19         for i in range(1,n):
20             #update F
21             F[i]=min(n2,n3,n5)
22
23             #update the pointers to l2, l3, and l5
24             if(F[i]==n2):
25                 cur2+=1

```

```

26         n2=F[cur2]*2
27     if(F[i]==n3):
28         cur3+=1
29         n3=F[cur3]*3
30     if(F[i]==n5):
31         cur5+=1
32         n5=F[cur5]*5
33     return F[-1]

```

6.1.4 Super Ugly Number (LeetCode 313)

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k.

Example:

```

Input: n = 12, primes = [2,7,13,19]
Output: 32
Explanation: [1,2,4,7,8,13,14,16,19,26,28,32] is the
    ↪ sequence of the first 12
        super ugly numbers given primes =
    ↪ [2,7,13,19] of size 4.

```

Note:

1. 1 is a super ugly number for any given primes.
2. The given numbers in primes are in ascending order.
3. $0 < k \leq 100$, $0 < n \leq 10^6$, $0 < \text{primes}[i] < 1000$.
4. The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

Our solution is treat this problem as the extension of the problem **Ugly Number II**. This method's time complexity is $O(nk)$, where k is the length of the array primes. But this method is not optimized, which should be speeded up to $O(n \log k)$. Think about the problem **Merging k Sorted Lists**, which is optimized by using the data structure heap.

```

1 class Solution(object):
2     ...

```

```

3      2018-12-25
4      '''
5      def nthSuperUglyNumber(self, n, primes):
6          """
7              :type n: int
8              :type primes: List[int]
9              :rtype: int
10             """
11             F=[1]*n
12             pointers=[0]*len(primes)
13             for i in range(1,n):
14                 #merge the lists
15                 F[i]=min([primes[j]*F[pointers[j]] for j in
16                     ↪ range(len(primes))])
17                 #update pointers
18                 for j in range(len(primes)):
19                     if(F[i]==primes[j]*F[pointers[j]]):
20                         pointers[j]+=1
21             return F[-1]

```

6.2 Permutation

6.2.1 Next Permutation (LeetCode 31)

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place and use only constant extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

```

1,2,3 -> 1,3,2
3,2,1 -> 1,2,3
1,1,5 -> 1,5,1

```