Algorithms

December 27, 2018

Contents

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.

```python
#Definition for singly-linked list.
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class List(object):
    def __init__(self, array):
        if(array):
            self.head = ListNode(array[0])
            prev=self.head
            cur = None
            for i in range(1,len(array)):
                cur=ListNode(array[i])
                prev.next=cur
                prev=cur
        else:
            self.head = None

    def __str__(self):
        if(not self.head):
            return "[]"
        else:
            s="["
            cur=self.head
            while(cur):
                s=s+str(cur.val)+" "
                cur=cur.next
            s=s.strip()+"]"
            return s


array1=[1,4,5]
list1=List(array1)
print(list1)
```

2 Sliding Widow 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):
                d.pop(first)
            else:
                d[first]-=1
```

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

```
            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 $floor((idx - 1)/2)$.

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,
            ↪  then pop out the max
28          if(not self.array):
29              raise ValueError("pop out from an empty heap")
30
31
            ↪  self.array[0],self.array[-1]=self.array[-1],self.array[0]
32          max=self.array.pop()
33          self.siftdown(0)
34          return max
35
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.

```
import heapq
'''2018-12-24
Use python's heapq to implement a binary max heap.
'''

class MaxHeapObj(object):
    def __init__(self, val):
        self.val = val

    def __lt__(self, other):
        return self.val > other.val

    def __eq__(self, other):
        return self.val == other.val

    def __str__(self):
        return str(self.val)


class MaxHeap(object):
  def __init__(self):
      self.h = []

  def heappush(self,x):
      heapq.heappush(self.h,MaxHeapObj(x))

  def heappop(self):
      return heapq.heappop(self.h).val

  def __getitem__(self,i):
      return self.h[i].val

  def __str__(self):
      s=""
      for e in self.h:
          s=s+str(e)+" "
      return s

```

```
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=""
```

```
        for i in range(len(self.h)):
            s=s+str(self[i])+" "
        return s


heap=MaxHeap()

for i in range(1,10):
    heap.heappush(i)

print("After inserting 1,2,3,4,5,6,7,8,9, the array of the heap is {0}.".format(heap))

maxInHeap=heap.heappop()
print("Pop out from the heap, we'll get the maximum number {0}, "
      "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 complexity[3].

[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
```

```python
import heapq

# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

class List(object):
    def __init__(self,array):
        if(array):
            self.head = ListNode(array[0])
            prev=self.head
            cur = None
            for i in range(1,len(array)):
                cur=ListNode(array[i])
                prev.next=cur
                prev=cur
        else:
            self.head = None

class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        heap=[]
        #init heap, which contains the tuple of head.val and
        ↪  head of each list
        for list in lists:
```

```
            if(list):
                heapq.heappush(heap,(list.val,list))

        dummy=ListNode(0)
        tail_new_list=dummy
        #update
        while(heap):
            _,head=heapq.heappop(heap)
            if(head.next):
                heapq.heappush(heap,(head.next.val,head.next))

            tail_new_list.next=head
            tail_new_list=tail_new_list.next
        return dummy.next

def display(head):
    s=""
    cur=head
    while(cur):
        s+=str(cur.val)+" "
        cur=cur.next
    return s


array1=[1,4,5]
list1=List(array1)
print(list1)

array2=[1,3,4]
list2=List(array2)
print(list2)

array3=[2,6]
list3=List(array3)
print(list3)


solution=Solution()
mergedList=solution.mergeKLists([list1.head,list2.head,list3.head])
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/

```
        """
        :type k: int
        :type nums: List[int]
        """
        #self.array is the min heap
        self.heap=nums
        heapq.heapify(self.heap)
        self.k=k
        while(len(self.heap)>k):
            heapq.heappop(self.heap)


    def add(self, val):
        """
        :type val: int
        :rtype: int
        """
        if(len(self.heap)<self.k):
            heapq.heappush(self.heap,val)
        else:
            if(val>self.heap[0]):
                #update min heap
                heapq.heappop(self.heap)
                heapq.heappush(self.heap,val)
        return self.heap[0]

# Your KthLargest object will be instantiated and called as
↪  such:
# obj = KthLargest(k, nums)
# 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 $O(n \log n)$.

```python
import heapq
import collections

class Solution(object):
    '''
    2018-12-27
    min heap method
    '''
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        #firstly, count the frequent.
        #The time complexity is O(n).
        numfreq=collections.defaultdict(int)
        for num in nums:
            numfreq[num]+=1

        #secondly, use the min heap with
        #the fixed size k to get the top k frequent
```

```
        #numbers. The time complexity is O(n * log k)
        heap=[]
        for num in numfreq:
            freq=numfreq[num]
            if(len(heap)<k):
                heapq.heappush(heap,(freq,num))
            else:
                if(freq>heap[0][0]):
                    heapq.heappop(heap)
                    heapq.heappush(heap,(freq,num))

        #thirdly, output the result from the min heap.
        #The time complexity is O(k * log k).
        res=[]
        while(heap):
            num=heapq.heappop(heap)[1]
            res.append(num)
        res.reverse()
        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 $O(n \log n)$.

```
import heapq
import collections

class Solution(object):
    '''
    2018-12-27
    max heap method
    '''
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
```

```
        :type k: int
        :rtype: List[int]
        """
        #firstly, count the frequent.
        #The time complexity is O(n).
        numfreq=collections.defaultdict(int)
        for num in nums:
            numfreq[num]+=1

        #secondly, use the max heap with
        #the size n to heapify the frequent-number tuples
        #The time complexity is O(n)
        heap=[(-1*freq,num) for num,freq in numfreq.items()]
        heapq.heapify(heap)

        #thirdly, pop out the top k frequent elements.
        #The time complexity is O(k*log n)
        res=[]
        while(len(res)<k):
            num=heapq.heappop(heap)[1]
            res.append(num)
        return res
```

4 Dynamic Programming

5 Maths

5.1 Prime numbers

5.1.1 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 $[1, \cdots, n]$ is a prime or not.

```
1  import math
2
3  class Solution(object):
4      '''2018-12-15
5      This is an O(n logn) 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
           ↪  number.
15         primeFlags=[True]*(n+1)
16         primeFlags[0]=False
17         primeFlags[1]=False
18
19         for p in range(2,int(math.sqrt(n))+1):
20             if(primeFlags[p]==True):
21                 for multiplier in range(p,n//p+1):
22                     primeFlags[p*multiplier]=False
23         return sum(primeFlags[:-1])
24
25     def test(self):
26         print("n={0}, output={1},
           ↪  expected={2}".format(5,self.countPrimes(5),2))
27         print("n={0}, output={1},
           ↪  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.

5.1.2 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).

```
class Solution(object):
    '''2018-12-25
    '''
    def isUgly(self, num):
        """
        :type num: int
        :rtype: bool
        """
        if (num<1):
            return False

        if (num == 1):
            return True

        while (num > 1):
            ugly = False
            for p in [2, 3, 5]:
                if (num % p == 0):
                    num = num / p
                    ugly = True
                    break
            if (ugly == False):
                return False
        return True


a=Solution()
print(a.isUgly(2147483648))
```

The recursion version is as following.

```
class Solution(object):
    '''2018-12-25
    '''
    def isUgly(self, num):
        """
        :type num: int
        :rtype: bool
        """
        def helper(num):
            if(num<=0):
```

```
                return False
            if(num==1):
                return True

            if(num%2==0):
                return helper(num//2)
            if(num%3==0):
                return helper(num//3)
            if(num%5==0):
                return helper(num//5)
            return False

        return helper(num)
```

5.1.3 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 $F = [1] + merge(l_2, l_3, l_5)$. 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.

```
class Solution(object):
    '''
    2018-12-25
    '''
    def nthUglyNumber(self, n):
        """
        :type n: int
        :rtype: int
        """
        F=[0]*(n+1)#F is the list of ugly numbers
        F[0]=0
        F[1]=1
        l2=[1*2]#l2 is the list of 2*F
        l3=[1*3]#l3 is the list of 3*F
        l5=[1*5]#l5 is the list of 5*F

        cur2=0
        cur3=0
        cur5=0
        for i in range(2,n+1):
            #update F, since F is the merge of l2, l3, and l5
            ↪  except for 0, and 1.
            #So we apply the merging method to update F.
            #And l2, l3, and l5 is based on F, so the update
            ↪  is simutaneously,
            #very tricky.
            F[i]=min(l2[cur2],l3[cur3],l5[cur5])

            #update l2, l3, and l5
            l2.append(F[i] * 2)
            l3.append(F[i] * 3)
            l5.append(F[i] * 5)

            #update the pointers to l2, l3, and l5
            if(F[i]==l2[cur2]):
                cur2+=1
            if(F[i]==l3[cur3]):
```

```
                cur3+=1
            if(F[i]==l5[cur5]):
                cur5+=1
        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.

```
class Solution(object):
    '''
    2018-12-25
    '''
    def nthUglyNumber(self, n):
        """
        :type n: int
        :rtype: int
        """
        F=[0]*n
        F[0]=1
        n2=2 #the value of current node in l2
        n3=3 #the value of current node in l3
        n5=5 #the value of current node in l5

        cur2=0 #the pointer to the current node in l2
        cur3=0 #the pointer to the current node in l3
        cur5=0 #the pointer to the current node in l5
        for i in range(1,n):
            #update F
            F[i]=min(n2,n3,n5)

            #update the pointers to l2, l3, and l5
            if(F[i]==n2):
                cur2+=1
                n2=F[cur2]*2
            if(F[i]==n3):
                cur3+=1
                n3=F[cur3]*3
            if(F[i]==n5):
                cur5+=1
```

```
                n5=F[cur5]*5
        return F[-1]
```

5.1.4 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. $0 < k \leq 100$, $0 < n \leq 10^6$, $0 <$ 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 $O(nlogk)$. Think about the problem **Merging k Sorted Lists**, which is optimized by using the data structure heap.

```
class Solution(object):
    '''
    2018-12-25
    '''
    def nthSuperUglyNumber(self, n, primes):
        """
        :type n: int
        :type primes: List[int]
```

```
        :rtype: int
        """
        F=[1]*n
        pointers=[0]*len(primes)
        for i in range(1,n):
            #merge the lists
            F[i]=min([primes[j]*F[pointers[j]] for j in
            ↪  range(len(primes))])
            #update pointers
            for j in range(len(primes)):
                if(F[i]==primes[j]*F[pointers[j]]):
                    pointers[j]+=1
        return F[-1]
```