

# Algorithms

December 25, 2018

## 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 <sup>1</sup>.

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

```

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

```

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

    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 <sup>2</sup>.

```
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.

---

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

```

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

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity<sup>3</sup>.

<sup>3</sup><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))

```

## 4 Dynamic Programming

### 4.1 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: Dynamic Programming.

We use the tabulation dynamic programming to compute n-th ugly number based on the previous ugly numbers. The recursive formula is defined as following.

$$F(n) = \begin{cases} 1, n = 1 & (a) \\ 2, n = 2 & (b) \\ 3, n = 3 & (c) \\ 4, n = 4 & (d) \\ 5, n = 5 & (e) \\ \min\{2 * F(n - 3), 3 * F(n - 3), 5 * F(n - 3)\}, n > 5 & (f) \end{cases} \quad (1)$$

## 5 Maths

### 5.1 Prime numbers

### 5.2 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     '''
6     def countPrimes(self, n):
7         """
8         :type n: int
9         :rtype: int
10        """
11        if(n<=1):
12            return 0
13        #primeFlags[i]=True means the number i is a prime
14        ↪ 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(2,n//p+1):
22                    primeFlags[p*multiplier]=False
23        return sum(primeFlags[:-1])
24
25 def test(self):
26     print("n={0}, output={1},
27     ↪ expected={2}".format(5,self.countPrimes(5),2))
28     print("n={0}, output={1},
29     ↪ expected={2}".format(10,self.countPrimes(10),4))
```

```
28 | a=Solution()  
29 | 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[2*p:n:p] =  
↪ [False]*len(primeFlags[2*p:n:p])
```

### 5.2.1 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)
```