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.

```
#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
```

[1] 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 [2].

```
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 .

```python
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.

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

```python
class MaxHeap():
    '''
    2018-12-24
    The root is bigger than its left child and right child.
    '''
    def __init__(self):
        self.array=[]

    def insert(self,num):
        if(not self.array):
            self.array.append(num)
        else:
            self.array.append(num)
            self.siftup(len(self.array)-1)


    def siftup(self,idx):
        if(idx==0):
            return
        parentIdx=(idx - 1) // 2
        if(self.array[parentIdx]<self.array[idx]):
            self.array[parentIdx], self.array[idx] = \
                    self.array[idx], self.array[parentIdx]
            return self.siftup(parentIdx)

    def extractMax(self):
        #swap the head (max) and the last one in self.array, then pop out the max
        if(not self.array):
            raise ValueError("pop out from an empty heap")

        self.array[0],self.array[-1]=self.array[-1],self.array[0]
        max=self.array.pop()
        self.siftdown(0)
        return max


    def siftdown(self,idx):
        '''
```

```
        move the current node down
        '''
        if(not self.array or len(self.array)==1):
            return
        left=idx*2+1
        right=idx*2+2
        maxIdx=idx
        if(left<len(self.array) and self.array[maxIdx]<self.array[left]):
            maxIdx=left
        if(right<len(self.array) and self.array[maxIdx]<self.array[right]):
            maxIdx=right

        self.array[idx], self.array[maxIdx] = self.array[maxIdx], self.array[idx]
        #sift down the smaller number recursively
        if(idx!=maxIdx):
            self.siftdown(maxIdx)


    def __str__(self):
        s=""
        for i in range(len(self.array)):
            if(i!=len(self.array)-1):
                s+=str(self.array[i])+" "
            else:
                s+=str(self.array[i])
        return s


heap=MaxHeap()
for i in range(1,10):
    heap.insert(i)
print("After inserting 1,2,3,4,5,6,7,8,9, the array of the heap is {0}.".format(heap))

maxInHeap=heap.extractMax()
print("Pop out from the heap, we'll get the maximum number {0}, "
```

```
        "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

```

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

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

```python
import heapq
'''2018-12-24
Use python's heapq to implement a binary max heap.
Since heapq is the min heap, so we need to reimplement
MaxHeap by multiplying -1 to the item in min heap, and
multiply -1 as well when using heappop() funciton.
'''

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

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

    def heappop(self):
        return heapq.heappop(self.h)*(-1)
```

```python

    def __getitem__(self, i):
        return -1*self.h[i]

    def __str__(self):
        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))
```

4 Dynamic Programming

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

```python
import math

class Solution(object):
    '''2018-12-15
    This is an O(n logn) solution.
    '''
    def countPrimes(self, n):
        """
        :type n: int
        :rtype: int
        """
        if(n<=1):
            return 0
        #primeFlags[i]=True means the number i is a prime
        ↪  number.
        primeFlags=[True]*(n+1)
        primeFlags[0]=False
        primeFlags[1]=False

        for p in range(2,int(math.sqrt(n))+1):
            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.

5.2.1 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).

```python
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
```

```python


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

The recursion version is as following.

```python
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)
```

5.3 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
```

```python
        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]
```

5.4 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. $0 < k \le 100$, $0 < n \le 10^6$, $0 < \text{primes[i]} < 1000$.
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 $O(nlogk)$. Think about the problem **Merging k Sorted Lists**, which is optimized by using the data structure heap.

```
class Solution(object):
    '''
    2018-12-25
    '''
    def 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]
```