Algorithms

December 24, 2018

1 Sliding Widow Technique

1.1 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))
```

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

```
        #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()
```

1.2 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 .

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

2 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}, "
```

```
74        "and the array of the heap becomes
          ↪  {1}.".format(maxInHeap,heap))
```

2.1 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
```

```
  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**.

```
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)
```

```
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 .
```