Topic 2: Two Pointers and Sliding Window

Competitive Programming I (Fall 2019)

Ninghui Li (Purdue University)

Two Pointers

- One maintains two pointers (i.e., indices) into either one array or two arrays, and move the pointers based on values they point to.

- Simple yet powerful technique. Can be used in surprising ways.

LeetCode 167. Two Sum II - Input array is sorted

Given an array of integers that is already *sorted in ascending order*, find two numbers such that they add up to a specific target number. You may assume that each input would have *exactly* one solution and you may not use the *same* element twice.

```
Input: numbers = [1,2,7,9,11,15], target = 11

Output:                [2,  4]
Explanation: The sum of 2 (at index 2) and 9 (at index 4) is 11.
This LeetCode question uses indices starting at 1 instead of 0.
```

Do we need to consider all pairs of numbers?

Two Sum II: Using Two Pointers

```
1 public int[] twoSum(int[] nums, int target) {
2   int first = 0,  second = nums.length-1;
3   while (/* ??? */) {
4     int sum = nums[first] + nums[second];
5     if (sum > target)           second--;
6     else if (sum < target)      first++;
7     else return new int[]{first+1, second+1};
8   }
9   return null;
10}
```

Time complexity is linear because each iteration updates exactly one pointer by one. Hence the loop body is executed less than N=num.length times. And the loop body execute a few operations.

- What goes on line 3? What is the time complexity?

Line 3 should be while (first<second). Because the two pointers move towards center, and cannot use the same element twice (hence first=second not acceptable.)

LeetCode 11. Container With Most Water

Given *n* non-negative integers *a1*, *a2*, ..., *an* , where each represents a point at coordinate (*i*, *ai*). *n* vertical lines are drawn such that the two endpoints of line *i* is at (*i*, *ai*) and (*i*, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note: You may not slant the container and *n* is at least 2.

```
Input: [1,3,8,6,9,8,3,6,3,4]
```

```
Output: 30
Explanation: the largest container can be formed using 8 at index
2, and 6 at index 7, which can hold (7-2)*min(6,8)=30.
```

Container with Most Water: Take One

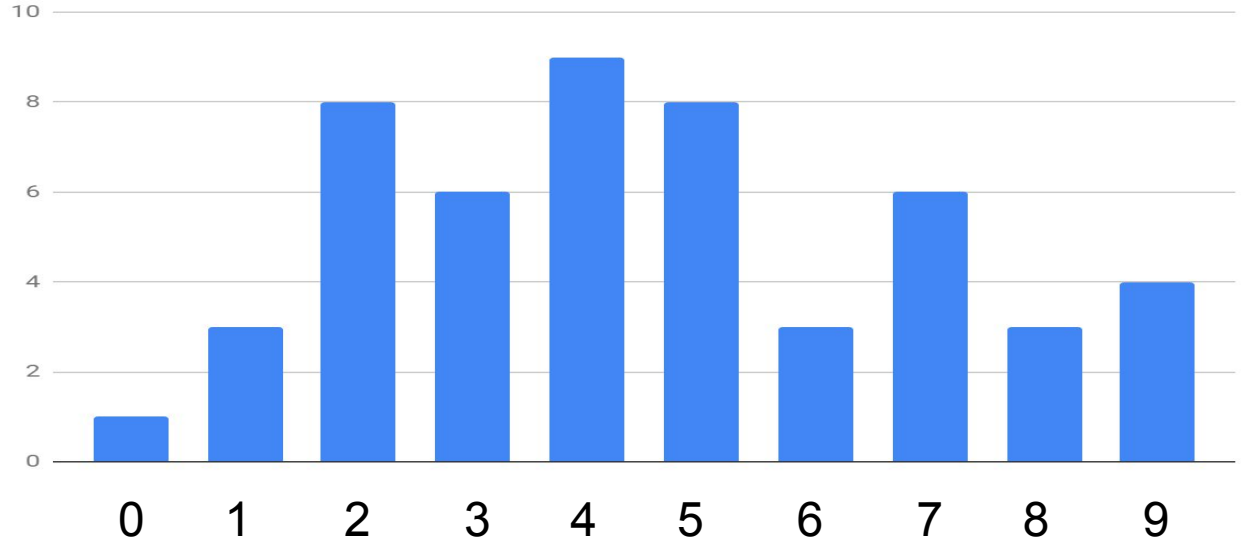
```
public int maxArea(int[] h) {
  int ans = Integer.MIN_VALUE,  N = h.length;
  for (int i=0; i<N; i++)  for (int j=i+1; j<N; j++) {
    ans = Math.max(ans, (j-i) * Math.min(h[i], h[j]));
  }
  return ans;
}
```

- What is the time complexity of this approach?
- A: O(N) B: O(N^2) C: O(N^3)

Complexity is O(N^2) because loop body is executed N(N-1)/2 times, and each execution is constant time.

Can we improve the time complexity? Do we need to check all pairs?

Container with Most Water: Illustration



Container with Most Water: Analysis

- How to improve the running time complexity?
- Can we skip checking some pairs?
- Key insight:
  - if (i<j && h[i] < h[j])
  - Any container [i,k] where i<k<j holds (min(h[i],h[k]) *(k-i)), and is always holds less than container [i,j] (holds h[i]*(j-i))
- Can be solved using two pointers, moving from both ends towards the middle.

Two Pointers Review

- One maintains two pointers (i.e., indices) into either one array or two arrays, and move the pointers based on values they point to.
- When it works, the key idea is a form of search space pruning. One knows that certain search space does not contain solution, and can thus skip searching in that space.
- See USACO 2015 December Silver 2. High Card Wins for a slightly different application of two pointers.

Sliding Windows

- Let us figure out what they are through examples.

LeetCode 239. Sliding Window Maximum

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position. Return the sequence of max values of the sliding windows.

```
Input: nums = [1,3,-1,-3,5,3,6,7], and k = 3
Output: [3,3,5,5,6,7]
```

Sliding Window Maximum: Take One

```
1 class Solution {
2  public int[] maxSlidingWindow(int[] nums, int K) {
3    if (K > nums.length)          return null;
4    if (K == 0)                   return new int[0];
5    int N = nums.length;
6    int[] ans = new int[N-K+1];
7    for (int i=0; i<=N-K; i++) {
8      ans[i] = Integer.MIN_VALUE;
9      for (int j=i; j<i+K; j++)
10       ans[i] = Math.max(ans[i], nums[j]);
11   }
12   return ans;
13 }
14}
```

What is the time complexity?

- A: O(N)
- B: O(N^2)
- C: O(N K)
- D: O(N K^2)

Can we speed this up?

Time complexity is O(N K), when K<N, or O(N^2), when K is unknown. We have an outer-loop of (N-K) times, and an inner-loop of K times. Worst case is when K=N/2, where line 10 executes around (N^2)/4 times.

Sliding Window Maximum: Take Two (Lazy Update)

```
1.  public int[] maxSlidingWindow(int[] nums, int k) {
2.   if (k > nums.length)          return null;
3.   if (k == 0)                   return new int[0];
4.   int N = nums.length;
5.   int[] ans = new int[N-k+1];
6.   for (int i=0; i<N-k+1; i++) ans[i] = Integer.MIN_VALUE;
7.   for (int j=0;j<k;j++)   ans[0]=Math.max(ans[0],nums[j]);
8.   for (int i=0; i<N-k; i++) {
9.    /* condition one for knowing max w/o scanning  */
10.   /* condition two for knowing max w/o scanning  */
11.   else for(int j=i+1; j<=i+k; j++)
12.           ans[i+1] = Math.max(ans[i+1], nums[j]);
13.  }
14.  return ans;
15. }
```

```
9   if (nums[i+k]>=ans[i])    ans[i+1]=nums[i+k];
10  else if (nums[i]<ans[i]) ans[i+1]=ans[i];
```

PriorityQueue

A PriorityQueue is a collection that supports efficient add/remove element, and find the least element.
Here we want the largest, instead of least. So:

```
PriorityQueue<Integer>  q =
    new PriorityQueue<Integer>(k,
        Collections.reverseOrder());
```

The **reverseOrder()** is a Java Collections class method which returns a comparator that imposes the reverse of the natural ordering on the objects.

```
q.add(x);     // add x to the queue, O(log N)
q.remove(x); // remove x from queue, O(log N)
q.peek();     // gets least element,  O(1)
q.poll();     // gets and removes least element O(log N)
```

Take Three (Using PriorityQueue)

```
 1.  public int[] maxSlidingWindow(int[] nums, int k) {
 2.    ...
 3.  PriorityQueue<Integer>  q =
 4.        new PriorityQueue<Integer>(k,
 5.            Collections.reverseOrder());
 6.   for (int i=0; i<k; i++)      q.add(nums[i]);
 7.   res[0] = q.peek();
 8.   for (int i=k; i<nums.length; i++) {
 9.    q.remove(nums[i-k]);
10.    q.add(nums[i]);
11.    res[i-k+1] = q.peek();
12.   }
13.   return res;
14.  }
```

PriorityQueue versus TreeSet/TreeMap

TreeSet has add(x), remove(x), first(), last() (All O(log N) time.)

Can one use TreeSet instead of PriorityQueue? What are their differences?

Differences between PriorityQueue (PQ) and TreeSet

- The same element can be added multiple times to PQ, but not to TreeSet.
  - However, one can use TreeMap to implement all functionalities for PQ. TreeMap maps each element to number of times it appears.
- TreeSet/TreeMap have functionalities that are not supported by PQ, such as iterating over all elements in order.
  - PQ has O(1) peek, although poll /remove/add take log(N), like TreeSet.

Conclusion: One can use TreeSet/TreeMap instead in Java.

Take Four (Using TreeMap)

```
 1.  public int[] maxSlidingWindow(int[] nums, int k) {
 2.    ...
 3.    TreeMap<Integer,Integer> q = new
         TreeMap<Integer,Integer>();
 4.    for (int i=0; i<k; i++) q.merge(nums[i],1,Integer::sum);
 5.    res[0] = q.lastKey();
 6.    for (int i=k; i<nums.length; i++) {
 7.     int v = q.get(nums[i-k]);
 8.     if (v > 1) q.put(nums[i-k], v-1);
 9.     else       q.remove(nums[i-k]);
10.     q.merge(nums[i], 1, Integer::sum);
11.     res[i-k+1] = q.lastKey();
12.    }
13.    return res;
14.  }
```

Sliding Window Maximum:

- Approach 1: Compute max of each sliding window
- Approach 2: Lazy: If cannot tell max of new window, compute max.
- Approach 3: Use a PriorityQueue to maintain elements in each window.
- Approach 4: Using TreeMap instead of PriorityQueue.
-
- Order the approaches from slowest to fastest?
- A: 1, 2, 3, 4 B: 1, 2, 4, 3 C: 4, 3, 1, 2
- D: 1, 4, 2, 3 E: 1, 3, 4, 2

Performance Comparison

	Brute Force	Laze Update	PriorityQueue	TreeMap
Time Complexity	O(NK)	O(NK)	O(N log K)	O(N log K)
Running Time	17 ms, faster than 30.52%	3 ms, faster than 94.54%	37 ms, faster than 22.58%	68 ms, faster than 6.66%
Memory Used	40.8 MB, less than 92.19%	42.3 MB, less than 43.75%	41.3 MB, less than 81.25%	41.8 MB, less than 59.38%

Results depend highly on test cases!
Under what condition(s) would PriotyQueue/TreeMap be better?
How to improve the lazy update to be faster?

USACO 2011 November Silver #2: Cow Lineup

Given N (up to 50000) cows, each standing at a location on a line and has a breed. (both up to 1 billion) FJ wants to take a photograph in which there is at least one cow of each distinct breed appearing. Compute the minimal size of the picture.

Input: 25 7 26 1 15 1 22 3 20 1 30 1

Output: 4 (A picture taken for 22 to 26 includes all three breed ids.)

Location	15	20	22	25	26	30
Breed ID	1	1	3	7	1	1

Code for Sorting 2D int Array

```
static void sort2DInt(int[][] array) {
  Comparator<int[]> comp = new Comparator<int[]>() {
   public int compare(int[] a, int[] b) {
     for (int i=0; i<a.length; i++) {
      if (a[i] < b[i])       return -1;
      else if (a[i] > b[i])  return 1;
     }
     return 0;
   }
  };  // Code for comparator can be copied when necessary
      // such as when using TreeSet<int[]>
  Arrays.sort(array, comp);
}
```

Cow Lineup: States of Sliding Windows Illustration

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

15	20	22	25	26	30
1	1	3	7	1	1

Cow Lineup: What to Maintain for a Window

- What state should we maintain for each window?
  - What do we need to know for each window?
    - Whether the window contains all distinct breeds!
  - We need the number of distinct breeds in a window.
  - How to update the number of distinct breeds efficiently?
    - We need a mapping from each breed id to the number of cows of that breed.
- Needs to remember how many cows of each breed are in the window, and the total number of different breeds.

Sliding Window Technique

- Applicable when there is a monotonicity property.
- There are two kinds of monotonicity property
  - If a window is good, then any window including it is also good.
  - If a window is good, then any sub-window is also good.
- Typically used to solve problems
  - Counting number of valid windows.
  - Find an optimal window according to some metrics of optimality.

Implementing Sliding Window: Three Approaches

1. Outer loop iterates through all values for the starting pointer, inner loop keeps increasing ending pointer as needed.
2. Outer loop is for each value for the ending pointer, inner loop keeps increasing starting pointer as needed.
3. A single loop, each iteration increases either the starting or the ending pointer by one.

What is the time complexity of such a sliding window algorithm (assuming moving one step takes constant time)?

Such a sliding window takes linea time, just like two-pointers. Each iteration, one of the beginning or ending shift by one.

Which of the three approaches should we use?

Sliding Window Approach 2: PseudoCode

When a problem wants Minimum Window (whole array always okay):

```
int start = 0;
for (int end=0; end<N; end ++) {
  // Add A[end] into the window
  // If window good, increase start while window still good,
  //   then check whether this is a better solution
}
```

When a problem wants Maximum Window (empty always okay):

```
int start = 0;
for (int end=0; end<N; end ++) {
  // Add A[end] into the window
  // If window bad, increase start until no longer bad
  // Check whether this is a better solution
}
```

Sliding Window Approach 3: PseudoCode

```
// When a problem wants Minimum Window (whole array always okay):
int start=0, end=-1;   // empty window.
while (true) {
  if (window good) {
    // check whether this gives a better solution
    // start++; update window to remove A[start]
  } else {
    // end++;
    // break if end is too large.
    // update window to include A[end]
  }
}
```

Cow Lineup Code, Using Approach 2

```
int[][] d;      // d stores sorted cows (loc,breed)
HashMap<Integer,Integer> bm = … ;   // maps all breeds to 0
int  B = bm.size(), start=0;
int  breeds = 0, ans=Integer.MAX_VALUE;
for (int end=0; end<d.length; end++) {
 if (bm.merge(d[end][1],1,Integer::sum) == 1) { // new breed
  if (++breeds == B) {  // covered all breeds if true
   while (bm.get(data[start][1]) > 1) // can move start
     bm.merge(d[start++][1],-1,Integer::sum);
   ans = Math.min(ans, d[end][0]-d[start][0]);
  }
 }
}
```

Implementing Sliding Window: Three Approaches

1. Outer loop iterates through all values for the starting pointer, inner loop keeps increasing ending pointer.
2. Outer loop is for each value for the ending pointer, inner loop keeps increasing starting pointer.
3. A single loop, each iteration increases either the starting or the ending pointer.

From my experience, one should always use Approach 2.

LeetCode 76. Minimum Window Substring

Given a string S and a string T, find the minimum window in S which contains all the characters in T in complexity O(n).

- If a character C appears in T k times, the window should contain at least k times of C.
- If there is no such window in S that covers all characters in T, return the empty string `""`.
- If there is such a window, you are guaranteed that there will always be only one unique minimum window in S.

```
Input: S = "ADOBECODEBANC", T = "ABC"
Output: "BANC"
```

Minimum Window Substring

- Need to figure out how many times each character in T occurs.
  - Can use Map<Character,Integer> or int[] (since characters should be ASCII, and between 0 and 127)
  - Again, an Array is a map.
- To enable fast check for whether a window in S contains all characters in T, maintain a counter of how many characters in T have been “satisfied”. (Similar to Cow Lineup.)

LeetCode 713. Subarray Product Less Than K

Given an array of positive integers `nums`. Count the number of (contiguous) subarrays where the product of all the elements in the subarray is less than `k`.

```
Input: nums = [10, 5, 2, 6], k = 30
```

```
Output: 6
```

Cannot use Approach 3 because we want to count number of valid windows.

Number of Valid Windows Using Approach 2

```
int start = 0, count=0, product=1;
for (int end=0; end<N; end ++) {
  product *= nums[end];       // Add A[end] into the window
  while (product >= k)  {
    product /= nums[start];
    start++;
  }
  count += end - start + 1;
}
```

Let us code if we have time.