

Topic 1: Prefix Sums

Competitive Programming I
Ninghui Li (Purdue University)

Example: LeetCode 724 Find Pivot Index

Given an array of integers `nums`, the pivot index is the index where the sum of the numbers to the left of the index is equal to the sum of the numbers to the right of the index.

Compute the pivot index. If none exists, return -1. If multiple exist, return the left-most one.

Input: `nums = [1, 7, 3, 6, 5, 6]`

Output:

3



Find Pivot Index: Take One

```
1 public int pivotIndex(int[] nums) {  
2     int N = nums.length;  
3     for (int i=0; i<N; i++) {  
4         int lsum = 0,    rsum = 0;  
5         for (int j=0; j<i; j++)    lsum+=nums[j];  
6         for (int j=i+1; j<N; j++) rsum+=nums[j];  
7         if (lsum == rsum)    return i;  
8     }  
9     return -1;  
10 }
```

The loop body from line 4-7 is executed N times. The body contains two loops, together they perform $C \cdot N$ operations, for some small constant C. Thus the time complexity is $O(N^2)$, meaning that the code requires $C \cdot N^2$ clock cycles for some C.

1. What is the time complexity of this approach?

A: $O(N)$ B: $O(N^2)$
C: $O(N^3)$ D: Unsure
E: Don't understand the question

2. What computations are repeated?

3. How to avoid repetition?

Find Pivot Index: Take Two

- Uses two scans over the array, with time $O(N)$, and $O(1)$ extra space
- First scan computes **total sum**
- Second scan computes **prefix sums**

```
1. public int pivotIndex(int[] nums) {  
2.     int sum = 0, psum=0;  
3.     for (int x : nums)    sum += x;  
4.     for (int i=0; i<nums.length; i++) {  
5.         if (/* ??? */) return i;  
6.         psum += nums[i];  
7.     }  
8.     return -1;  
9. }
```

5: if
(psum+psum+nums[i]==sum)

Line 3 has complexity $O(N)$.
The loop body line 5-6 is
executed N times. The body
performs a few operations.
Thus the time complexity is
 $O(N)$.

Lesson from “Find Pivot Index”: Prefix Sums

Given an array A of n elements,

The prefix sums are defined as:

$$S_0 = 0,$$

$$S_j = (A[0] + \dots + A[j-1]) \quad \text{for } j \text{ in } [1..n].$$

There are $n+1$ prefix sum values for an array of size n !

Using prefix sums can sometimes turn quadratic approach into linear!

Example: [LeetCode 974 Subarray Sums Divisible by K](#)

Given an array `A` of integers, return the number of (contiguous, non-empty) subarrays that have a sum divisible by `K`.

`1 <= A.length <= 30000`

`-10000 <= A[i] <= 10000`

`2 <= K <= 10000`

Input: `A = [4,5,0,-2,-3,1], K = 5`

Output: 7

Explanation: There are 7 subarrays with a sum divisible by `K = 5`:

`[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3],`

`[0], [0, -2, -3], [-2, -3]`

[Discuss](#)

Subarray Sums Divisible by K: Take One (Bruteforce)

```
1 public int divByK(int[] A,int K){
2   int count = 0, N = A.length;
3   for (int i=0; i<N; i++)
4     for (int j=i; j<N; j++) {
5       int sum = 0;
6       for (int p=i; p<=j; p++)
7         sum += A[p];
8       if (sum % K == 0)
9         count++;
10  }
11  return count;
12 }
```

What is the time complexity of this approach?

A: $O(N)$ B: $O(N^2)$ C: $O(N^3)$
D: $O(N^2 K)$ E: $O(N K^2)$

What computation are repeated?

How to avoid them?

Main computation is lines 5-9, which is executed $O(N^2)$ times (about $N^2/2$ times). Here lines 6-7 is another loop, which executes $O(N)$ times (perhaps around $N/3$ on average). Overall complexity is thus $O(N^3)$.

Analysis: LeetCode 974 Subarray Sums Divisible by K

- The Naive algorithm is to check all subarrays $A[i..j]$, compute their sums, and check whether it works.
- There are $O(N^2)$ pairs to check, and $O(N)$ to compute sum, so total complexity is $O(N^3)$.
- Observation: The sum of a subarray $A[\text{range}(i,j)]$ (all elements from $A[i]$ to $A[j-1]$), equals the difference of prefix sums: $S_j - S_i$.
- If we store all prefix sums in an array, computing subarray sums takes constant time.

Subarray Sums Divisible by K: Take Two (Prefix-sum array)

```
1. public int subarraysDivByK(int[] A, int K) {
2.     int count = 0, N = A.length;
3.     int[] p = new int[N+1];    // prefix sum array
4.     for (int i=0; i<N; i++) ... // What code goes here?
5.     for (int i=0; i<N; i++) for (int j=i; j<N; j++) {
6.         if ((p[j+1]-p[i]) % K == 0) count++;
7.     }
8.     return count;
9. }
```

Line 4: $p[i+1] = p[i] + A[i]$;

- What is the time complexity of this approach?
- A: $O(N)$ B: $O(N^2)$ C: $O(N K)$ D: $O(N^2 K)$
- Can we speed it up further?

Main computation is on line 6, which is executed $O(N^2)$ times, and takes constant time. Overall complexity is thus $O(N^2)$.

Subarray Sums Divisible by K: Analysis Cont'd

Use a simple recursion-based formulation. Let us figure out how to compute the number of solutions ending at a given index.

A		4	5	0	-2	-3	1
Prefix sums	0	4	9	9	7	4	5
Prefix sums Mod 5	0	4	4	4	2	4	0
Solutions ending at each index			[5]	[0], [5,0]		[-2,-3] [0,-2,-3] [5,0,-2,-3]	Whole array
# solutions ending at index	0	0	1	2	0	3	1

How to efficiently compute # solutions ending at an index?

Subarray Sums Divisible by K: Analysis Cont'd

Given the array [4, 5, 0, -2, -3, 1]. Consider the following example.

Sum for the prefix array with one element (i.e., [4]) is 4, which has remainder (mod 5) being 4.

Sum for the prefix array with three elements (i.e., [4, 5, 0]) is 9, which also has remainder (mod 5) being 4.

That the two remainders are the same means that the difference of the two prefix sums, which is the sum of the subarray [5,0], is a multiple of $K=5$.

Thus the number of continuous subarrays that have a sum to be a multiple of K and ends at the index j equals the number of prefix sums that end before index j and have the same remainder mod K .

Subarray Sums Divisible by K: Analysis Cont'd

What matters are the number of times each remainder (of prefix sums modulo K) appear!

Approach 1 (linear time): Store how many time each possible remainder has appeared so far. **What data structure to use?**

- The number of subarrays ending an a given index j equals the number of times that $(S_j \bmod K)$ appears in prefixes before j .
- Use either an array or a HashMap to store how many times a remainder has appeared so far. Updating and looking up takes constant time.

Subarray Sums Divisible by K: Analysis Cont'd

Approach 2: Compute the number of times each remainder appears in total. Let P_r be the number of times that remainder r appears, then # of solutions is $\sum_{r=0,1,\dots,K-1} (P_r * (P_r - 1) / 2)$. (P_r chooses 2)

- E.g., in the example, 4 appears 4 times, 0 appears twice, hence result is $(4*3)/2 + (2*1)/2 = 7$.

Approach 3: Sort the array of remainders modulo K at the end, to compute how many times each remainder appears. Running time would be $(N \log N)$.

Lesson from LeetCode 974

- Any subarray sum can be computed as difference of two prefix sums
- To find multiples of K , one considers remainders modulo K
- The information of how many prefix sums have a given remainder can be stored in different ways; which is better depends on K
 - When K is small, uses an array, where $R[i]$ is the number of prefix sums with remaining mod K equalling i
 - When K is large, we use a Map.
- Thinking about counting how many solutions ending at an index is an important generic technique.

Set

- A set is a collection containing no duplicate elements
- Operations on sets include:
 - Testing for membership
 - Adding elements
 - Removing elements
 - Union
 - Intersection
 - Difference
 - Subset

Methods in The Java Set Interface

```
boolean contains (E e)      // member test
boolean add (E e)           // adding, enforces no-duplicates
boolean remove (Object o)
boolean isEmpty ()
int size ()
Iterator<E> iterator ()
boolean containsAll (Collection<E> c)  // subset test
boolean addAll  (Collection<E> c)      // set union
boolean removeAll (Collection<E> c)    // set difference
boolean retainAll (Collection<E> c)    // set intersection
```


Java Classes Implementing Set

HashSet

- Implemented using hashing (actually using HashMap underneath)
- Methods add/remove/contains/size takes $O(1)$ time

TreeSet

- Implemented using a balanced binary search tree
- Implement SortedSet
 - first() : Returns the first (lowest) element
 - last() : Returns the last (highest) element
 - headSet/tailSet/subSet (returns a portion of the set)
- Methods add/remove/contains/first/last takes $O(\log N)$ time

Map

- Map is a set of ordered pairs, each (*key*, *value*)
- In a given **Map**, there are no duplicate *keys*
- Each key is mapped to one value
- Different keys can be mapped to the same value
- Can think of key as “mapping to” a particular value
- An array is a Map from a set of integers to a set of values

Some Methods of java.util.Map<K, V>

```
// K is the key type
// V is the value type
V get (Object key)           // may return null
V put (K key, V value)       // returns previous value or null
V remove (Object key)        // returns previous value or null
boolean isEmpty ()
int size ()
containsKey(Object key)
Set<K>  keySet()
Set<Map.Entry<K,V>> entrySet()
```

Using Map:

- A helpful method: `p.merge(k, v, Function)` does the following

```
e = p.get(k) ;  
if (e==null) {  
    p.put(k,v) ;  
    return v;  
} else {  
    e=Function(e,v) ;  
    p.put(k,e) ;  
    return e;  
}
```

Java Classes Implementing Map

- HashTable (HashMap supporting multi-threaded access)
 - Not needed in competitive programming
- HashMap
 - (Amortized) Constant, i.e., $O(1)$, time for get/put/remove
- TreeMap
 - Implemented SortedMap
 - Have firstKey(), lastKey(), ...
 - $O(\log N)$ time for get/put/remove

Example: [LeetCode 560 Subarray Sum Equals K](#)

Given an array of integers and an integer k , find the total number of continuous subarrays whose sum equals to k .

The length of the array is in range $[1, 20,000]$. Range of numbers in the array is $[-1000, 1000]$ and the range of k is $[-1e7, 1e7]$.

Input: `nums = [1,1,1], k = 2`

Output: 2

Analysis: LeetCode 560 Subarray Sum Equals K

- Similar to previous problem (subarray sum multiple of K).
- But we want subarrays of $\text{sum} == K$, rather than sum multiples of K.
- We still need to compute prefix sums. How to use them?
- Given a prefix sum S , we want to check how many times the value $S-K$ appears before.
- How about we store prefix sums in an array, and then check?
 - Checking takes $O(N)$ time.
- What do we need?
 - Quickly check how many times a value exists. What to use?

Using Map

- We want to figure out for each value, how many time it has been added. We need a Map.
- Logically, each `int[]` array `A` is a map, that maps an integer `i` (index value) to an integer `A[i]` (element).
- When indices are sparse, using Array takes too much space.
- We can use `HashMap` or `TreeMap`.
 - Important methods: `put(k,v)`, `get(k)`, `size()`
 - For `HashMap`, these operations take constant time on average.
 - For `TreeMap`, `put/get` takes time $O(\log N)$, where `N` is the number of keys in the map, which is close to constant .

Debug Time: Find the Bugs in the Code.

```
1. public int subarraySum(int[] nums, int k) {
2.     HashMap<Integer,Integer> s=new HashMap<Integer,Integer>();
3.     int sum = 0, count = 0;
4.     for (int x : nums) {
5.         sum += x;
6.         count += s.get(k-sum);
7.         s.put(sum, 1);
8.     }
9.     return count;
10. }
```

Bugs:

Before line 4, should add `s.put(0,1);`

Line 6 should be something like

`Integer x = s.get(sum-k);`

`If (x != null) count += x;`

Line 7 should be

`s.merge(sum, 1, Integer::sum);`

Time complexity is linear, because line 5-7 are executed N times, and the map operations take constant time.

Kattis: [Baloni](#)

There are N ($1 \leq N \leq 1M$) balloons floating, lined up from left to right, each at height H_i , where $1 \leq H_i \leq 1M$. One wants to pop all of them by shooting arrows at chosen heights. Each arrow travels from left to right. When an arrow touches a balloon, the balloon pops and disappears and the arrow continues its way at a height decreased by 1. Output the minimal number of arrows that is needed.

Input:

5

4 5 2 1 4

Output:

3

Baloni: Illustration

5	Arrow 2					
4	Arrow 1					
3						
2	Arrow 3					
1						

Baloni: Analysis

One brute-force approach is to simulate effect of each arrow.

- Scanning from left to right, a new balloon requires a new arrow, follow the trace of this arrow, which may destroy other balloons. Repeat while there are balloons left. **What is the complexity of this approach?**
- Simulating one arrow takes $O(N)$ time, and we may need $O(N)$ arrows, for a complexity of $O(N^2)$


How to avoid scanning through all balloons many times?

Simulate multiple arrows at the same time! How to do that efficiently?

Use either a Map or a histogram array to store the state of multiple arrows (mapping height to number of arrows travelling at that height).

Example: LeetCode 287: Find the Duplicate Number

Given an array *nums* containing $n + 1$ integers where each integer is between 1 and n (inclusive). There is only one duplicate number in the array, **but it could be repeated more than once**. Find the duplicate number.



If the duplicate number appears only once, then a simple linear algorithm exists.

1. Your runtime complexity should be less than $O(n^2)$.
2. You must not modify the array (assume the array is read only).
3. You must use only constant, $O(1)$ extra space.

Here, we only consider how to solve this problem with two out of the three constraints above. (Solutions will be accepted on LeetCode without 2 or 3.)

Analysis: LeetCode 287: Find the Duplicate Number

1. Your runtime complexity should be less than $O(n^2)$.

Without this constraint, can do a double loop.

2. You must not modify the array (assume the array is read only).

Without this constraint, can first sort the array (`Arrays.sort(nums);`).
Then a linear scan.

3. You must use only constant, $O(1)$ extra space.

Without this constraint, store which elements one has seen while scanning. **What data structure to use for storing that?**