Competitive Programming

From Problem 2 Solution in O(1)

Two Pointers Technique

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

Two Pointers Technique

- Pointer = index
  - No relationship to C++ pointers **int **x;**
- It is not a specific algorithm. Just easy idea that might be effective for specific problems
- You probably coded it before, but don't know a name for it
- In 2010, with a Codeforces tag, the name become more popular
- I will utilize this tutorial

Two Pointers Technique

- Technique that uses 2 **constrained** indices (move of one **can** be limited by the another)
  - Typically each pointer iterates on O(N) array positions.
  - Hence overall **increment/decrement** is O(N)
- Applications
  - In sorted arrays, where we want to find some positions
  - Or cumulative array of positive numbers array (sorted)
  - Variable size sliding window, where we search for a window (range) of specific property (max sum)
  - Ad Hoc cases

Sum of 2 numbers problem

- It is one of the best problems to clarify the 2-pointers technique
- Given a **sorted** array A, having N integers. You need to find any **pair(i,j)** having sum as given number **X**.
  - O(N^2): 2 nested loops and compare the sum
  - O(Nlogn): For each array value V, binary search for X-V
  - O(N) using 2-pointers!

Sum of 2 numbers problem

- 2-pointers based on the sortedness of array
  - Let pointer(index) p1 on the first element of array
  - Let p2 on the last element of the array
  - Let Y = the sum of these 2 numbers
  - If Y > X => shift p2 to the left => decrease Y
  - If Y < X => shift p1 to the right => increase Y
  - Keep doing so untill Y == X or no way
  - Then each pointer moves O(N), total O(N)

Sum of 2 numbers problem

- Let A = {2, 4, 5, 7, 8, 20}, X = 11
  - P1 = 0, P2 = 5, Y = 2 + 20 = 22 > 11
  - The only thing we can do is to move **p2 left**
  - P1 = 0, P2 = 4, Y = 2 + 8 = 10 < 11
  - Now we need bigger sum => move **p1 right**
  - P1 = 1, P2 = 4, Y = 4 + 8 = 12 > 11
  - Again, move **p2 left** to decrease sum
  - P1 = 1, P2 = 3, Y = 4 + 7 = 11 == 11 (Found)

Sum of 2 numbers problem

```
#define lli long long

bool f(lli sum) {
    int l = 0, r = n - 1; //two pointers
    while ( l < r ) {
        if ( A[l] + A[r] == sum ) return 1;
        else if ( A[l] + A[r] > sum ) r--;
        else l++;
    }
    return 0;
}
```

Sliding Windows

- A window is a range with start/end indices
  - So by definition, we have a point for its start & end
  - **Fixed size window of length K**
  - In this windows, we have specific range and searching for a range with specific property. Easy to handle
  - **Variable size window**
  - In this windows, the window can be of any size. More tricky

Recall: Fixed size sliding window

- Given an array of N values, find M consecutive values that has the max sum?
- A brute force to compute that is just O(NM) by starting from every index and compute M values. Matter of 2 nested loops
- Observation: What is the relation between the first M values and 2nd M values?

Fixed size sliding window

- Let A = {1, 2, 3, 4, 5, 6, -3}, M = 4
  - 1st M values = **1**+2+3+4 = 10
  - 2nd M values = 2+3+4+**5** = 10-**1**+**5** = 14
  - 3rd M values = 3+4+5+6 = 14-2+6 = 18
  - 4th M values = 4+5+6-3 = 18-3-3 = 12
  - So answer is max(14, 18, 12) = 18
- We create a **window** of **fixed** size M
  - cur window = last window - its first item + new mth item
  - Window start = pointer 1
  - Window end = pointer 2
  - P2 = P1+K-1

Variable size sliding window

- Find a range with property
  - Given an array having N **positive** integers, find the contiguous subarray having sum as **max** as possible, **but** <= M
  - Let p1 = p2 = 0
  - Keep moving p2 as much as the window is **ok**
  - Once window is **!ok** = stop p2
  - keep moving p1 as long as window is !**ok**
  - Once window is **ok** = stop p1 and back to p2 again
  - For any **ok** window (here sum <= M), do your evaluations
  - Remember this strategy :)

Variable size sliding window

- Let A = {2, 4, 3, 9, **6, 3, 1**, 5}, M = 10
  - P1 = 0, P2 = 0, Y = 2 = 2 <= 10. P2++
  - P1 = 0, P2 = 1, Y = 2+4 = 6 <= 10. P2++
  - P1 = 0, P2 = 2, Y = 2+4+3 = 9 <= 10. P2++
  - P1 = 0, P2 = 3, Y = 2+4+3+9 = 18 > 10. P1++
  - P1 = 1, P2 = 3, Y = 4+3+9 = 16 > 10. P1++
  - P1 = 2, P2 = 3, Y = 3+9 = 12 > 10. P1++
  - P1 = 3, P2 = 3, Y = 9 = 9 <= 10. P2++
  - P1 = 3, P2 = 4, Y = 9+6 = 15 > 10. P1++
  - P1 = 4, P2 = 4, Y = 6 = 6 <= 10. P2++
  - P1 = 4, P2 = 5, Y = 6+3 = 9 <= 10. P2++
  - P1 = 4, P2 = 6, Y = 6+3+1 = 10 .. max stop

Variable size sliding window

```
int l = 0, r = 0;
    lli ans = 0;

while ( l < n ) {
   while ( r < n && sum + A[r] <= M ) {
       sum += A[r];
       r++;
   }
   ans = max(ans, sum);
   sum -= A[l];
   l++;
}
```

Variable size sliding window

- Another (critical) example
  - Given an array containing N integers, you need to find the length of the **smallest** contiguous subarray that contains at least K **distinct** elements in it.
  - As we said, P1=P2 = 0. Shift P2, then P1, P2, P1….etc
  - But what makes a window ok?
  - As long as we don't have k distinct numbers = **OK**
  - How to know current count?
  - Maintain a set & map datastructure of the current numbers
  - P2 adds its number, P1 remove its number

Variable size sliding window

```
int l = 0, r = 0, ans = INF;
map <int , int >  cnt;

while ( l < n ) {
    while ( r < n && s.size() < K ) {
       s.insert(A[r]);
       cnt[A[r]]++;
       r++;
    }
    if (s.size() >=K) {
                ans = min(ans, r-l);
        }
    if ( cnt[A[l]] == 1 ) s.erase(A[l]);
    cnt[A[l]]--;
    l++;
}
```

Your turn

- Given an array having N integers, find the contiguous subarray having sum as **max** as possible
  - Now 2 changes occurred.
  - Numbers can be +ve or -ve
  - We are not limited by a limit
  - What makes a window ok? When to P2++ or P1++?
  - This problem is know as Maximum Sum 1D

Your turn

- Given two sorted arrays A and B, each having length N and M respectively. Form a new sorted merged array having values of both the arrays in sorted format.
  - This is 2 arrays not just one! They are also sorted
  - Let P1 = 0 on Array A
  - Let P2 = 0 on Array B
  - Let C is the new array
  - What is C[0]? A[p1] or B[P2]?
  - This is an important step of the merge sort algorithm

Summary

- Examples summary
  - So we maintain 2 (or more?) pointers on an array
  - Case: p1 = start, p2 = end
  - Case: p1 = start, p2 = start+fixed_length
  - Case: p1 = start, p2 = start
  - Case: p1 = start of array, p2 = start of another array
- Some popular algorithms are related, explicitly or implicitly, to 2-pointers
  - Reverse string (We can do that with 2 points (0, n-1) and do swapping)
  - Quick sort, Mrege sort, Z-function, Prefix function

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً