Competitive Programming

Book - 101

Basic Codechef Understanding And Few Problems Understandings

→ Prime Seive

→ WA, Sgv error mean in code chef etc.

Efficient Coding With Example.

Problems.

#1 To print the elements in 2-D array in single loop only :-

n → Array length

⇒ It's however like converting $O(n^2)$ in $O(n+n)$ but it's not.

⇒ However not sure if its exactly $O(n+n)$ since printing all element is however is stating $O(n*n)$ only. We are only just making only use of single while loop.

e.g. code :-

```
int[][] arr = {{1,2,3,4},{5,6,7,8},{10,20,30,40}};
int i=0;
int j=0;

while(i<arr.length){
    Sysо(arr[i][j]);

    if(j<arr[i].length){
        j++;
    }

    if(j==arr[i].length){
        j++;
        j=0;
    }

}
```

⇒ But this still is $O(n^2)$

Output ⇒ 1 2 3 4 5 6 7 8 10 20 30 40

#(2) TLE in finding pairs in array such that

arr[i] - arr[j] = k for given k.

⇒ Given arr = [1, 2, 3, 4]

key = 1

out pairs ⇒ (2,1), (3,2), (4,3)

Hence output = 3.

⇒ In O(n²) which got stuck in TLE.

```
static int pairs (int k, int[] arr){
    Arrays.sort(arr);
    int count = 0;
    for(int i=0; i<arr.length-1; i++){
        for(int j=i+1; j<arr.length; j++){
            if(arr[j]-arr[i]==k){
                count++;
            }
        }
    }
}
```

⇒ Now in O(n) using Two pointer. → Best for ?

⇒ ~~[illegible]~~.

⇒ The following code got accepted. (You may have to sort Arrays first or may be not) And first you've to sort it

```
// First sort the array.
static int pairs (int k, int[] arr){
    int i=0; int j=1; int n=arr.length;
    int count=0;

    while(j<n){
        int diff = arr[j]-arr[i];
        if(diff==k){
            count++;
            j++;
        }
        else if(diff>k){
            i++;
        }
        else if(diff<k){
            j++;
        }
    }

    return count;
}
```

O(n log n)

Can be done by Set better way

⇒ Also there's another solution for this question
O(n) time complexity by Umesh using set.
Hashset finds element in O(1) technique used here.

⇒ It will work for sorted, unsorted, duplicates or not.

Code in C++; O(n) time complexity.

It's Java code next page see

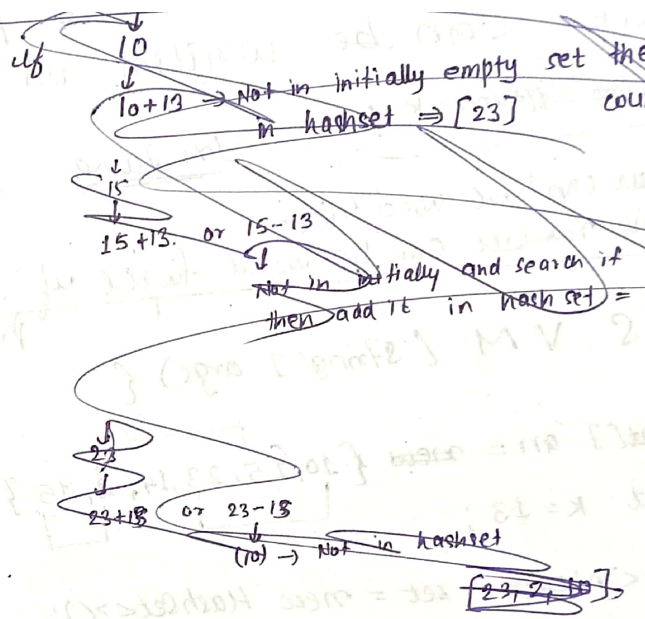
```
int main() {
    vector<int> arr = [10,15,23,14,2,15];
    int K=13;
    int n=arr.size(), count=0;
    set<int> s;
    for(int i=0; i<n; i++) {
        if(s.find(arr[i]+k) != s.end()) {
            count++;
        }
        else if(s.find(arr[i]-k) != s.end()){
            count++;
        }
        s.insert(arr[i]);
    }
    cout<<count;
    return 0;
}
```

Do in Java will do

write behind concept also

See next page.

output 3



Finally Now I understood it's work.

OK now understood

K=13

[10 15 23 14 2 15]

Not understood totally C++ stuff.

10 not in Hashset → add in hashset [10]

(10-13) or (10+13) Not Contain in Hash

count=0

Sel	count	i=	diff/sum
10	0	arr[0]=10	23, -3
[10,15]	0	arr[1]=15	2, 27
[10,15,23]	1	arr[2]=23	10(present) 33
[10,15,23,14]	1	arr[3]=14	27, 1
[10,15,23,14,2]	2	arr[4]=2	15 (repeat) 11
[10,15,23,14,15]	3	arr[5]=15	2, 27 (present)

But you might think (23) was there if we already have put all elements in Hashset, But it will take two count for (10,23) and (23,10) so it's better we take into count from 23 only and keep on steady adding. So understood now fully,

Simply in Java the previous code can be written as below:-

2) And no. of pairs whose difference = k :-

In Java

one pair can't be used twice
one number can be used twice if it is present twice

```
P S V M (String[] args) {

    int[] arr = new { 10, 15, 23, 14, 2, 15 };
    int k = 13;

    Set<Integer> set = new HashSet<>();
    int count = 0;

    for (int i=0 ; i<arr.length ; i++) {

        if (set.contains (arr[i]+k)) {
            count++;
        }
        else if (set.contains (arr[i] - k)) {
            count++;
        }
        set.add (arr[i]);
    }

    Syso (count);  // output = 3
}
```

i.e. 23 → 10 ⇒ 23-13 ⇒ 10
2 → 15 → 2+13 → 15
15 → 2 → 15-13 ⇒ 2

③ BigInteger 'OR' method to do bitwise 'OR' operation.

e.g

```
BigInteger bigInteger = new BigInteger ("2300");
BigInteger val = new BigInteger ("3400");

BigInteger ans = bigInteger.or(val);

Syso(ans); // 3580
```

Binary of 2300 = 100011111100
Binary of 3400 = 110101001000

OR of 2300, 3400 → 110111111100 → 3580.

⇒ Similarly we have

① and() method
② xor()
③ not()
④ getLowestSetBit().

Application of this OR in BigInteger in PES

② Hackerrank ACM ICPC Team

Given string[] arr ⇒ input ⇒

	member 1	2	3	4	5
1 →	1	0	1	0	1
2 →	1	1	1	1	0
3 →	0	0	0	1	1

Adding or ORing any two we want max possible 1's.

e.g.

Members (combination of two from)	Subjects	
(1,2)	[1,2,3,4,5]	⇒ max topic a pair team can know. = ⑤.
(1,3)	[1,3,9,5]	no. of such pair = ①
(2,3)	[1,2,3,9]	

So output 5,1

Code :-

```
static int[] acmTeam (String[] topic) {
    BigInteger[] bi = new BigInteger[n];
    int n = topic.length;

    for (int i = 0; i<n; i++) {
        bi[i] = new BigInteger (topic[i], 2);
    }
    int maxTopic = 0;
    int teamCount=0;

    for (int i=0; i< n; i++) {
        for (int j = i+1; j<n; j++){
            BigInteger iuj = bi[i].or(bi[j]);
            int bitCount = iuj.bitCount();
            if (bitCount > maxTopic) {
                maxTopic = bitCount;
                teamCount =1;
            } else if (bitCount == maxTopic) {
                teamCount++;
            }
        }
    }
```

```
    int[] result = {maxTopic, teamCount};
    return result;
}
```

So for input :-

```
4 5
10101
11100
11010
00101
```

→ output

```
5
2
```

(But it's $O(n^2)$ soln.) check if it worked in Hackerrank.

explanation :-

(1,2) → 4
(1,3) → 5
(1,4) → 3
(2,3) → 4
(2,4) → 4
(3,4) → 5

The 2 team (1,3) & (3,4) know all 5 topics which is maximal.

#4 Window Sliding Technique :-

⇒ This technique shows how a nested for loop in few problems can be converted to single for loop and hence reducing the time complexity.

⇒ Problem statement :-

Given an array of integeres of size 'n'. Our aim is to calculate the maximum sum of 'k' consecutive elements in the coreay.

input[] arr ⇒ arr[] = {100, 200, 300, 400}

K = 2

output = 700.

So it's for K consecutive elements or K subarray elements

Since consecutive elements of size k so why to sort it first so

we don't have to do sorting first to apply sliding window technique if we want consecutive

⇒ using brute-force our code will be as :-

```
static int maxSum (int arr[], int n, int k) {
    int max_sum = Integer.MIN_VALUE;

    for(int i=0; i<n-k+1; i++) {
        int current_sum = 0;
        for(int j=0; j<k; j++) {
            current_sum = current_sum + arr[i+j];
        }
        max_sum = Math.max(current_sum, max_sum);
    }
    return max_sum;
}
```

⇒ The above code time-complexity is O(k*n) as it contains 2 nested loops.

Window sliding Technique :-

for sub Array → Mean K consecutive elements

consider an array arr[] = {5, 2, -1, 0, 3}

and K = 3 , n = 5 .

our initial window

our next window and so on.

add subt.

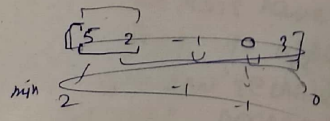
current_sum = window_sum

current_sum = window_sum + 0 - 5

So in this window just add the next element and remove (subtract) the 1st element of window and so on. it goes in single loop.

→ But first we've to create this window and then later use this window in our single for loop.

code as.



→ code as :- using

```
static int maxSum (int arr[], int n, int k){
    // k must be greater
    if(n<k){
        Syso("Invalid");
        return -1;
    }

    //compute sum of first window of size k
    int max_sum = 0;
    for (int i=0; i<k; i++){
        max_sum += arr[i];
    }

    //compute sums of remaining windows by removing first element of
    // previous window and adding last element of current window
    int window_sum = max_sum;
    for (int i=k; i<n; i++){
        window_sum += arr[i] - arr[i-k];
        max_sum = Math.max (max_sum, window_sum);
    }
    return max_sum;
}
```

→ Here Time complexity is O(n)

#5 Two Pointers Technique :-

(Array must be sorted)

⇒ Two pointers is an easy and effective technique which is typically used for searching pairs in a sorted array.

⇒ Problem statement :- Given a sorted array A (sorted in ascending order), having N integers, find if there exists any pair of elements (A[i], A[j]) such that their sum is equal to X.

⇒ Lets see naive brute force approach. Time = $O(n^2)$

```
static boolean isPairSum (int A[], int N, int X){
    for (int i=0; i<N; i++){
        for (int j=0; j<N; j++){
            if (A[i]+A[j] ==X){
                return true;
            }
            if (A[i] + A[j] >X){
                break;   //as the arr is sorted.
            }
        }
    }
    return false;
}
```

⇒ Now by two pointer. one for first element other for last element in the array. we add the values kept at both the pointer.

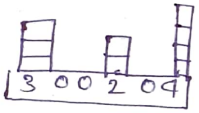
If their sum is smaller than X then we shift the left pointer to right (since sorted in ascending) or if their sum is greater than X then we shift the right pointer to left, in order to get closer to the sum. we keep moving the pointer until we get the sum as X. (Also this can be used to count pairs whose sum is X)

code as :

```
static boolean isPairSum (int A[], int N, int X) {
    int i = 0;
    int j = N-1;
    while (i<j) {
        if (A[i] + A[j] == X){
            return true;        count++; (in case we want to find number of pairs whose sum is X.
        }
        else if (A[i] + A[j] < X){
            i++;
        }
        else {
            j++;
        }
    }
    return false;
}
```

#6 Trapping Rain Water :- (Some optimized technique)

⇒ This tells a way to use techniques to do in O(n) rather than O(n²).



Total trapped water = 3+3+1+3 = 10.

⇒ A simple solution is to traverse every array element and find the highest bars on left and right sides. Take the smaller of two heights. The difference between the smaller height and height of the current element is the amount of water that can be stored in this array element.
Time complexity = O(n²).

⇒ An efficient solution is to pre-compute highest bar on left and right of every bar in O(n) time. Then use these pre-computed values to find the amount of water in every array element. Time complexity = O(n).

⇒ code as :-

```
static int findWater(int n){

    int left[] = new int[n];

    //left[i] contains the height of tallest bar to the left of
    //ith bar including itself.

    //right[i] contains height of tallest bar to the right of
    // ith bar including itself.
    int right[] = new int[n];

    int water = 0;

    //fill left array.
    left[0] = arr[0];
    for(int i = 1; i<n; i++){
        left[i] = Math.max(left[i-1], arr[i]);
    }

    //fill right array:
    right[n-1] = arr[n-1];
    for(int i = n-2; i>=0; i--){
        right[i] = Math.max(right[i+1], arr[i]);
    }

    //calculate the accumulated water element by element: consider the water amount on ith bar, the
    // amount of water accumulated on this particular bar will be equal to min(left[i],right[i])-arr[
    for(int i=0; i<n; i++){
        water += Math.min(left[i],right[i]) - arr[i];
    }
    return water;
}
```