Binary Search Problems - Medium - Hard

Created	@July 20, 2022 8:00 PM
Class	
Type	
Materials	
Reviewed	☐

Problem 1:

https://codeforces.com/contest/1324/problem/D

Given two arrays each of length n (n ≤ 2*10^5). Both the arrays represent the value of degree of interesting topic that can be taught in some class. First array (A) represents how much the topic is interesting for the teacher and the second array (B) represents how much the topic is interesting for the students. Ai represents how much the ith topic is interesting for teacher, and Bi represents how much the ith topic is interesting for students.

We call a pair of topic to be **good** pair, if (Ai + Aj) > (Bi + Bj) for (i < j) i.e. the pair of topic is more interesting for the teacher.

We need to count all possible good pairs of topics.

Ex:

A = [4,8,2,6,2] // teacher → i = 1, j = 2 → Ai = 8 Aj = 2 → Ai+Aj = 10

B = [4,5,4,1,3] // students → Bi = 5, Bj = 4 → Bi + Bj = 9 → 10 > 9

Ans: 7

All the good pairs are (0,1) (0,3) (1,2) (1,3) (1,4) (3,4) (2,3)

Hint 1: The problem can be solved with merge sort as well as binary search. And there are few advanced methods as well (we will not discuss)

Hint 2: Can we simplify the equation ?

Hint 3: You might have solved the problem of counting inversions in an array. Can we use it here ?

Solution:

We can easily solve this problem using Binary Search.

Let's try to simplify the equation

```
Ai + Aj > Bi + Bj
Ai - Bi > Bj - Aj
Ai - Bi > -(Aj - Bj)
Assume Ai - Bi as Ci
Ci > -Cj
```

From the above simplification we can say, we need to find all j's for a given i such that j > i and

`Ci > -Cj`

Let's say we can create a new array C, such that Ci = Ai - Bi, and then sort the array C in ascending order. Because sorting the array will help us to apply binary search. But why do we want to apply binary search ?

We need to find all indexes j such that `j > i` and `-Ci < Cj` (we converted the inequality by multiplying -1 on both sides), so for any value Ci, we need to count the number of values which are greater than negative of Ci.

We know how to get the first value greater than a given value in a sorter array → `upper_bound` Upper bound gives us the first index greater than a value x.

So in the array we need to find all values greater than -Ci, after the index i. We can use upper bound to get the first value greater than -Ci, because if we get this first value then all the values to the right of it are also going to be greater as the array is sorted.

Example:

A = [4,8,2,6,2]

B = [4,5,4,1,3]

C = [0,3,-2,5,-1] // Ci = Ai - Bi

now sort the array C in ascending order

C = [-2,-1,0,3,5]

now for i = 0, we need to find all `j > i` such that `-Ci < Cj`

C0 = -2 → -C0 = 2, so from index `1 to C.length - 1` we need to find all values greater than 2

Upper bound of 2 in C is index 3, so everything starting from index 3 to the last index will all be greater than -C0. For i = 0, we got `n-upper_bound(-C0)` → `n-upper_bound(2)` → `n-3` → `5-3` → 2

So for i = 0, we got 2 pairs.

Now for i = 1, we need to find all `j > i` such that `-Ci < Cj`

C1 = -1 → -C1 = 1, so from index `2 to C.length - 1` we need to find all values greater than 1

Upper bound of 1 in C is index 3, so again we got 2 more pairs.

For i = 2, we need to find all `j > i` such that `-Ci < Cj`

C2 = 0, -C2 = 0 , upper bound of 0 from index 3 to C.length - 1 is 3, and again we got 2 more pairs.

for i = 3, C3 = 3, -C3 = -3, so from index `4 to C.length - 1`

we need to find all values greater than -3 , we have only 1 value, so one more pair.

That's how we got 7 pairs.

What about the time complexity ?

`O(nlogn)` why ? Because for all the values almost we are doing binary search on the remaining array. Total N values, and binary search of logn on each values leads to nlogn.

Space: `O(n)` → due to array C.

```
function upperBound(arr, x, st) {
  let start = st, end = arr.length - 1;
  let ans = arr.length;
  while(start <= end) {
    let mid = Math.floor(start + (end - start) / 2);
    if(arr[mid] <= x) {
      // discard left half
      start = mid + 1;
    } else {
      // element at mid can be a potential answer
      ans = mid;
      end = mid - 1; // go and find something even better on left side
    }
  }
  return ans;
}

let a = [4,8,2,6,2]
let b = [4,5,4,1,3]
let c = []
for(let i = 0; i < a.length; i++) {
  c.push(a[i] - b[i]);
}
c.sort((a, b) => a-b);
let ans = 0;
for(let i = 0; i < c.length - 1; i++) {
  let ub = upperBound(c, -c[i], i+1);
  ans += c.length - ub;
}
console.log(ans);
```

Problem 2:

SPOJ.com - Problem NOTATRI

You have N (3 ≤ N ≤ 2,000) wooden sticks, which are labeled from 1 to N. The i-th stick has a length of L i (1 ≤ L i ≤ 1,000,000). Your friend has challenged you to a simple game: you will pick three

https://www.spoj.com/problems/NOTATRI/

Given an array of integers of length n (n ≤ 2000) we need to find the count of triplets such that the triplets cannot form a triangle.

Example: [5,2,9,6]

Ans: 2 → (5,2,9) (2,9,6) these two triplets will not form a triangle.

Pre requisite: In order to solve the problem you need to have basic knowledge about the property of a triangle that is what combination of sides form a valid triangle. If you have three sides as a, b, c

```
1.a + b > c
2.a + c > b
3.b + c > a
```

All of the above three conditions should be true.

Hint 1: It is almost similar to the previous problem

Hint 2: A triangle will not be formed if any one of the 3 condition fails, i.e. for any pair a, b if we have a `c` such that a + b < c then this triplet will not form a triangle

Solution:

Brute Force: In the brute force approach we can try to form all possible triplets and then check what all triplets don't follow the property of a triangle.

This approach will require `O(n^3)`

```
for(let i = 0; i < n; i++)
  for(let j = i+1; j < n; j++)
    for(let k = j+1; k < n; k++)
      // console.log(a[i], a[j], a[k]) -> filter first and then print
```

How can we optimise ?

In order to find a triplet that doesn't form a triangle, we can try to find all the triplets (a,b,c) that follows the given condition i.e. `a+b < c`

If we somehow generate all possible `a,b` then we can try to find all the `c` such that `a+b < c`

To generate all possible `a,b` we can try to get all possible pairs in O(n^2) time. Now for a given pair, if we need to find all the values greater than it's sum, then if we have the access to the upper_bound then all the values from upper_bound are going to be greater than it.

[5,2,9,6,11] → [2,5,6,9,11]

let's say a = 2, b = 5, a+b = 7 → upper_bound will be index 3 → from here we got 2 triplet

let's say a = 2, b = 6, a+b = 8 → upper_bound will be index 3 → from here we got 2 triplet

So in the prev problem we were getting upper_bound of -Ci, instead of this we have to get upper bound of a+b in this question where a,b are all possible pairs.

Time Complexity: `O(n*n*logn)` → n^2 due to generation of all possible pairs and then for each pair logn for the upper bound.

Space: `O(1)`

Problem 3:

https://www.spoj.com/problems/ABCDEF/

https://www.spoj.com/problems/ABCDEF/

You are given an array of length n (n ≤ 100), we need to count all possible sextuples which follow the given equation

```
((a*b+c)/d ) - e = f // d != 0
```

Note: In a sextuple for this given question numbers can be repeated.

Example: [2,3] → one of the sextuples can be (a=2,b=3,c=2,d=2,e=2,f=2)

Ans: 4 , total 4 different sextuples will be there

Solution:

Brute force: In the brute force we can try to find all possible sextuples and then filter out the required one.

Time Complexity : `O(n^6)` This solution will give TLE

How can we optimise ?

```
// lets simplify the equation
((a*b+c)/d ) - e = f
((a*b+c)/d ) = f + e
(a*b+c) = (f + e)*d // d!=0
```

What is this simplified form denoting ?

The meaning is if we are able to find two triplets (a,b,c) and (d,e,f) such that they follow the above equation or we can say, if we put them in the above equation the values are equal then, these two triplets can be combined to form a sextuple.

Ex: (2,3,2) (2,2,2) → (2,3,2,2,2,2)

2*3 + 2 = 8

(2+2)*2 = 8

And, n can be as big as 100, so if we want to generate all possible triplets complexity will be `O(n^3)` so generating triplets will be within the bounds.

So, let's try to generate all possible triplets, and store them in two different arrays, such that in the first array we will compute them with LHS and in the second array we will compute them with RHS.

For example: [2,3] →

```
let arr = [3,2]; // sample code to print all triplets
for(let i = 0; i < arr.length; i++) {
    for(let j = 0; j < arr.length; j++) {
        for(let k = 0; k < arr.length; k++) {
            console.log("(" , arr[i], arr[j], arr[k], ")");
        }
    }
}
```

(a b c)

(3 2 3)
(3 2 2)
(2 3 3)
(2 3 2)
(2 2 3)
(2 2 2)

(3 3 3)
(3 3 2)

We will now have two arrays lhs and rhs, in lhs instead of putting the triplet directly, I will put the value of the triplet that we get by putting it in the lhs equation `a*b+c`

lhs = [9, 8, 9, 8, 7, 6, 12, 11]

And similarly we will create rhs array and instead of putting the triplets we will put the value that we get by putting the triplets in the rhs equation `(f + e)*d`

rhs = [15, 10, 15, 10, 12, 8, 18, 12]

(3,2,2) + (2,2,2) ⇒ (3,2,2,2,2,2)

(2 3 2) + (2,2,2) ⇒ (2,3,2,2,2,2)

(3,3,3) + (2,2,3) ⇒ (3,3,3,2,2,3)

(3,3,3) + (3 3 2) ⇒ (3 3 3 3 3 2)

The above 4 sex tuples are formed by getting common values from both lhs and rhs, because these common values denote that there are two triplets which satisfy our simplified equation.

So all we have to do is find the total common elements in two arrays.

We can iterate on any one say LHS, and for each value of lhs we will try to find the number of occurrences of the value.

How to find the occurrences efficiently ?

Let’s sort both of them, and then for each element of LHS we will apply lower bound and upper bound i.e. binary search on the rhs lhs = [9, 8, 9, 8, 7, 6, 12, 11] → [6,7,8,8,9,9,11,12]

rhs = [15, 10, 15, 10, 12, 8, 18, 12] → [8,10,10,12,12,15,15,18]

let’s do for 12, so lower bound of 12 in rhs is index 3, and upper bound is index 5. Total occurrences of 12 is ub - lb ⇒ 5-3 ⇒ 2

lb → first index ≥ x

ub → first index > x

```
function lowerBound(arr, ele) {
    /**
     * Time: O(logn)
     * Space: O(1)
     */
    let start = 0;
    let end = arr.length - 1;
    let ans = -1;
    while(start <= end) {
        let mid = start + Math.floor((end - start)/2);
        if(arr[mid] >= ele) {
            // then the current index can be one of our possible ans
            ans = mid;
            end = mid - 1;
        } else {
            start = mid + 1;
        }
    }
    return ans;
}
function upperBound(arr, ele) {
    /**
     * Time: O(logn)
     * Space: O(1)
     */
    let start = 0, end = arr.length - 1;
    let ans = arr.length;
    while(start <= end) {
        let mid = start + Math.floor((end - start)/2);
        if(arr[mid] <= ele) {
            // if the mid element is <= element, then jump right
            start = mid + 1;
        } else {
            // we found a poteintial ans, try to find better one on left side
            ans = mid;
            end = mid - 1;
        }
    }
    return ans;
}
```

```
let arr = [5,7,10];
let lhs = [], rhs = [];
for(let i = 0; i < arr.length; i++) {
    for(let j = 0; j < arr.length; j++) {
        for(let k = 0; k < arr.length; k++) {
            lhs.push((arr[i]*arr[j]) + arr[k]);
            if(arr[i] != 0) {
                rhs.push((arr[i])*(arr[j] + arr[k]));
            }
        }
    }
}

lhs.sort((a, b) => a-b);
rhs.sort((a, b) => a-b);
let ans = 0;
for(let i = 0; i < lhs.length; i++) {
    let ub = upperBound(rhs, lhs[i]);
    let lb = lowerBound(rhs, lhs[i]);
    ans += (ub - lb);
}
console.log(ans);
/**
Time: O(n*n*n*logn)
Space: O(n*n*n)
*/
```

Problem 4:

SPOJ.com - Problem SUBSUMS

Given a sequence of N (1 ≤ N ≤ 34) numbers S 1, ..., S N (-20,000,000 ≤ S i ≤ 20,000,000), determine how many subsets of S (including the empty one) have a sum between A and B

https://www.spoj.com/problems/SUBSUMS/

We are given an array of length n (n ≤ 34), we need to count the number of subsets whose sum lie in the range A, B.

Solution:

Brute Force:

In the brute force solution we can try to generate all possible subsets. O(2^n) → TLE

How can we optimise ?

If the value of N would have been lesser then the brute force solution was efficient enough to get submitted.

We cannot generate the subsets of the whole array, but we can think in the manner that what if, we divide our array into two halves, (in the worst case one half will be of size 17) And generate all possible subsets of each half separately, then can we do something ?

Ex: [1,2,3,4]

[1,2] → [], [1], [2], [1,2] → [0,1,2,3]

[3,4]→ [], [3], [4], [3,4]→ [0,3,4,7]

So to generate subsets of first half, in the worst case we need 2^17 iterations and for the second half also in the worst case we need 2^17 iterations.

Total iterations will be 2^17 + 2^17 = 2*(2^17) ⇒ 2*131072 ⇒ 262144 (This is way less than 10^8)

But due to this we missed some subsets like [2,3,4] [1,2,3,4] [2,4] [1,4] etc. So how can we use the subsets generated by the two halves so that we can consider all the subsets whose sum lie in the range A, B.

Can we use binary search here ?

Let's say instead of storing the subsets, we will store their sum. Now we have two array X and Y which have got sum of subsets of first and second half of the array.

Assume A = 0, B = 6 → there are some subsets like [1], [2], [1,2] whose sum is ≥ 0 and ≤ 6 and are coming just form the first half. There are some subsets like [3] [4] whose sum ≥ 0 and ≤ 6 and they are coming just from the second half.

But there will some subsets which can come by combining the two halves ? [1,3] [1,4] [2,3] [2,4] [1,2,3]

So, it's easy to find the subsets whose sum is ≥A and ≤B and are coming from just first or second

half. But how to calculate the combined subsets whose sum is ≥ A and ≤B.

Let's say we consider some value `v1` from the first half subset sum, to find which subset in the second half `v2` can combine with the subset with sum `v1` so that sum lies in the given range can we say that

```
A <= v1 + v2 <= B
// and we already fixed v1 form the left half subsets so
A - v1 <= v2 <= B - v1
```

and to find the subsets with a sum ≥ A - v1 can we use lowerBound ? and sum ≤ B - v1 can we use upperBound on the second half array.

[1,2] → [], [1], [2], [1,2] → X [0,1,2,3]

[3,4]→ [], [3], [4], [3,4]→ Y [0,3,4,7]

A = 0, B = 6

Iteration 0

v1 = 0 → [] empty subset

A - v1 = 0, B - v1 = 6

lowerBound(0) → index 0 and upperBound(6) → index 3 (for upperBound we have to see values to the left of it, why ? because we can sum to be less than or equal to B-v1, upperBound gives us first value greater than B-v1, so everything to left of upper bound will be valid)

Iteration 1

v1 = 1

A - v1 = -1, B - v1 = 5

lowerBound(-1) → index 0 and upperBound(5) → index 3

.

.

.

So for each element of `X` calculate lowerBound and UpperBound in `Y`.

Time complexity: O(2^(n/2) * log(2^(n/2))) → O((n) * 2^(n/2))

```
let arr = [1, -2, 3];
let A = -1, B = 2;
const left = arr.slice(0, arr.length/2);
const right = arr.slice(arr.length/2, arr.length);
const m = subsets(left); // implement subsets function
const n = subsets(right);

m.sort((a, b) => a-b);
n.sort((a, b) => a-b);
let ans = 0;
for(let i = 0; i < m.length; i++) {
    let lb = lowerBound(n, A-m[i]); // implement lowerbound
    let ub = upperBound(n, B-m[i]); // implement upperbound
    ans += (ub - lb);
}
console.log(ans);
```

Problem 5:

Given a number x, calculate integer part of square root of x.

Ex: x = 16 → ans = 4

x = 40 → ans = 6

x = 99 → and = 9

Solution:

Brute Force:

We know that sqrt of x will be some value less than x only, so we can iterate from 1 to x-1 and for each value in this range check if value*value ≤ x or not, the last value satisfying the condition will be the answer.

```
function sqrt(x){
  // Time: O(x)
  let ans = 0;
  for(let i = 1; i < x; i++ ){
    if(i*i <= x) ans = i;
    else break;
  }
  return ans;
}
```

How to optimise ?

Now we know that our ans lies in the range [1, x-1] always, so we can assume this as a search space where we will find the last `m` such that `m*m <= x`

Can we divide the search space based on some property such that the two halves are different from each other based on this property.

If we have a mid value, and `mid*mid > x` then, everything to the right of mid will also be greater than x, but everything to the left may or may not be greater.

Similarly, if `mid*mid <= x` then everything to the left of mid will satisfy this condition but not everyone on the right side.

Based on this property we can divide our search space, hence we can apply binary search.

```
function sqrt(x) {
/**
  Time: O(logx) Space: O(1)
*/
  let lo = 1, hi = x-1;
  let ans = -1;
  while(lo <= hi) {
    let mid = lo + Math.floor((hi - lo) / 2);
    if(mid*mid == x) return mid;
    if(mid * mid <= x) {
      ans = mid;
      lo = mid + 1;
    } else {
      hi = mid - 1;
    }
  }
  return ans;
}
```

Problem 7:

What if in the previous problem we have to calculate precision also ? Let's say we want ans upto 3 decimal places ?

Ex: x = 99 → ans 9.949

So, this time we want ans upto 3 decimal places.

Just consider for a moment, of wanted to find the answer only for 1 decimal place, then what were the candidate values, if you already know the integer part ?

`sqrt(99)` → integer part is 9 using binary search

For only 1 decimal place possible values were → `9.0, 9.1, 9.2, 9.3, 9.4 ..... 9.8, 9.9`

So the answer for 1 decimal place will be the last value from the above 10 values such that `value * value ≤ x` . So to get this last value we can do linear or binary search. Now using binary search here can be an overkill, because search space size is just 10, so both linear and binary search are gonna be constant. So we can just use linear search.

Now, if someone asks us for 2 decimal places, and we already have the answer of 1 decimal place i.e. 9.9, then the possible values will be `9.90, 9.91, 9.92, 9.93, 9.94 ..... 9.98, 9.99` again only 10 possible values can be there, so one by one check every value and get the last value such that `value*value <= x.`

So for the second decimal spot, `9.94` is the last value. And again here we have just a search space of 10 size, so just linear or binary search anything is doable.

Then do the same thing for 3rd decimal place, then for 4th place and so on and so forth.

```
Time: O(logn + k) -> logn for getting integer part, and K for the precision part because i
n each iteration for decimal part we do constant work and total iterations will be k.
```

> These type of problem where we don't have a dedicated search space like array and instead we try to denote the search space as all possible values of our answer, and then apply binary search is called as `Binary Search On Answer`

homework : write the code for handling the precision