PASCAL △


```
                1          → 1
              1   1        → 2
            1   2   1      → 3
          1   3   3   1    → 4
        1   4   6   4   1  → 5
      1   5  10  10   5   1 → 6
```

o) Given Row & column, Print the element at that place.

$R = 5 \quad C = 3$

ans = 6

o) Print any N^{th} row of pascal △

o) Given N, Print the entire △

① → ${}^{R-1}C_{C-1}$ if $\underline{R=5,\ C=3}$ then ${}^{4}C_2 = \dfrac{4 \times 3 \times 2 \times 1}{(2 \times 1)(2 \times 1)}$

$= 6$

$\therefore {}^{n}C_r = \dfrac{n!}{r! \times (n-r)!}$

∴ to avoid calculating these factorials like we used to do

${}^{7}C_2 = \left(\dfrac{7 \times 6}{2 \times 1}\right)$

$\dfrac{7 \times 6}{2!}$ → choice

order ← order do not matter

$^{10}C_3 = \left(\frac{10 \times 9 \times 8}{3 \times 2 \times 1}\right)$

but in loop calculate like $\left(\frac{10}{1}\right) \times \frac{9}{2} \times \frac{8}{3}$

if taken $\left(\frac{10}{3}\right)$ pos

```
~~for~~ (
fun NCR (n, r) {
    result = 1

    for (i = 0 ; i < r , i++) {
        result = res x (n - i);
        result = result / (i + 1);
    }
return result ;
}

fun NCR (n-1, c-1);
```

O(R)

(ii) Print row

Nth row → n elements / row BRUTE

∴
```
for (c = 1 ; c ≤ n ; c++) {
    Print ( fun NCR (n-1, c-1));
}
```

time - O(N x R)

(iii) → [[1]
[1 1]
[1 2 1]
[1 3 3 1]]

return in list format.

```
generateRow ( int row ) {
    List ansRow ;
    ansRow.add (1) ;
    for ( int i = 1 ⟶ row ) {
            ans = ans x (row - col ) ;
            ans = ans / col ;
            ansRow.add ( (int) ans ) ;
        }
    return ansRow ;
    }
Triangle ( int n ) {
List ans ;
for ( int row = 1 ⟶ = n ) {
            ans.add ( generateRow (row) ) ;  }
        return ans ;
    }
```

OPTIMAL

Now 6-row

1 5 10 10 5 1

$$ans = 1 \quad \frac{5}{1} \quad \frac{5}{1}\times\frac{4}{2} \quad \frac{5\times4}{1\times2}\times\frac{3}{3} \quad \frac{5\times4\times3}{1\times2\times3}\times\frac{2}{4} \quad \frac{5\times4\times3\times2}{1\times2\times3\times4}\times\frac{1}{5}$$

$^5C_0 \quad ^5C_1 \quad ^5C_2 \quad ^5C_3$ similarly ∴ addition of patter element col in pattern.

$\rightarrow 1 \quad \frac{5}{1} \quad \left(ans \times \frac{N-col}{col}\right)$ ∴ $1 \times \frac{6-1}{1} = 1 \times \frac{5}{1}$

formula

```
ans = 1
print (ans);
for (i=1 ⟶ n-1) {
for ( i = 1 ⟶ = (n-1)) {
        ans = ans x ((n - i) / (i))
        print (ans);
}
print
```

row

O(N)

MAJORITY ELEMENT

elements greater than $\left(\frac{n}{3}\right)$ if $\frac{8}{3} = 2$

arr = [1, 1, 1, 3, 3, 2, 2, 2]

BRUTE

find element count if $> \frac{n}{3}$ then store

also OBSERVATION that a the answer cannot have > 2 element as ex $n = \frac{33}{3} = 11 + 1 = 12$ as we want $> \frac{n}{3}$

$\therefore$ $12 + 12 = 24 + 12 = 36 \rightarrow$ greater $> n$

$\therefore$ at max only [-, -] two element in 0 element

$\therefore$ when we have two elements break

```
ls = []
for (i=0 → n) {
    if( ls.size ==0 || ls[0]!= nums[i]) {
        count=0;
        for (j=0; →n-1) {
            if (nums[j] == nums[i]) count++;
        }
        if(count > n/3) ls.add(nums[i]);
    }
    if( ls.size() == 2) break;
}
```

$O(n^2)$

space → $O(1)$ as only 2 elements

BETTER

- Take hashmap 

traverse and increase count:

but ~~only access map~~ store element as answer when the minimum criteria is fulfilled $\left(\left(\frac{n}{3}\right)+1\right)$ times

∴

```
ls = [ ]; map ;    int mini = (n/3)+1

for (i=0 → n){                                  O(n)
        int value = mpp.getOrDefault (arr[i],0);
        mpp.put (arr[i], value+1 );

        if (map.get (arr[i]) == min){           O(1)
                ls.add (arr[i]);                if unordered
        }
        if (ls.size() == 2) break;
}
```

space - O(n) when every element is different.

OPTIMAL

```
count1 = 0 ;   count2 = 0
ele1 ; ele2 ;

for(i=0 → n-1){
    if(count1==0 && nums[i] != ele2){
        count1 = 1 ,
        ele1 = nums[i];
    }
    else if(count2==0 && nums[i] != ele1){
        count2 = 1;
        ele2 = nums[i];
    }
    else if(ele1 == nums[i]) count1++;
    else if(ele2 == nums[i]) count2++;
    else {
        count1--;
        count2--;
    }
}
```

So that it does not count for element which is stored by (ele2)

O(n)

O(n)

when given an array that do no contain majority element or only 1 maj

at last iterate through array and check if ele1 and ele2 $> \frac{n}{3}$

arr = [2, 1, 1, 3, 1, 4, 5, 6]

cnt1 = 0 1 0 1 0 1

ele = 2 1 (6) $< \frac{n}{3}$ ×

cnt2 = 0 1 2 1 2 1

ele = 1 (1) —

Now

- Let $n = 33$ ∴ $\frac{n}{3} = \frac{33}{3} = (11)$
- majority when $= (11+1) = (12)$
- if two maj then $= 12+12 = (24)$

remaining element = 33 - 24 = 9

Now 9 element cannot cancel out 12 element of (mj)

in case where there is only one majority element the count of other element get to 0 which leads to if initialization and reinitialization.

time - O(2)

space - O(1)

3SUM

find tuplets that add up to 0. their index needs to different for every element. Do not give duplicate tuplets

BRUTE

- make all combination

```
arr = [-1, 0, 1, 2, -1, -4]
Set<List<Integer>> st = new HashSet<>();
for(i=0 -> n) {
    for(j=i+1 -> n) {
        for(k=j+1 -> n) {
            if(arr[j] + arr[i] + arr[k] == 0) {
                List<Integer> temp = Arrays.asList(arr[i], arr[j],
                                                   arr[k]);
                temp.sort(null);
                st.add(temp)
            }
        }
    }
}
return (new ArrayList<>(st));
```

$O(n^3)$

sorting so that it can be compared element by element

using set so there are no duplicate

if ordered set

time → $O(n^3 \times \log(\text{no of triplets}))$

space → $2 \times O(\text{no of triplets})$

~~OPTH~~ BETTER → $n^3 \to n^2$

arr = [-1, 0, 1, 2, -1, -4]

we will make pair of two and look for third in hashmap.

arr = [-1, 0, 1, 2, -1, -4]

i j j

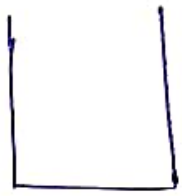
arr[k] = -(arr[i] + arr[j])

= ~~-(-1 + (-4))~~

= 5

= -(-1 + 0)

arr[k] = 1



not in set

∴ add j

↓

Now

arr[k] = -(-1 + 1)

= 0

in set

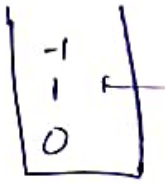
∴

triplet found

also add arr[j] in map

arr[k] = -(-1 + 2)

= -1



not in set then add

what we are doing is that fixing (i) then moving j if k in set triplet otherwise

[-1 0 1 2 -1, -4]

(i) (j)

all the elements will be in set to find the third one

[-1 0 1 2 -1 -4]

(-i) (j)

then empty set move i repeat.

```
set s ←
for (i=0 → n) {
    hashset h;
    for (j=i+1 → n) {
        int third = -(arr[i] + arr[j]);
        if (hashset.contains(third)) {
            List temp = Arrays.asList(arr[i], arr[j], third);
            temp.sort();
            st.add(temp);
        }
        hashset.add(arr[j]);
    }
}
return new ArrayList<>(st);
```

set to avoid duplicate

for finding if set is ordered otherwise O(1) in unordered

time - $O(n^2 \times \log M)$

space - $O(n) + O(\text{no of triplets}) \times 2$

when storing max element in hashset on first pass

[-1 0 1 2 -1 -4]

(-i) (j)

then empty set move i repeat.

```
set i ←
for (i=0 → n) {
    hashset j
    for (j=i+1 → n) {
        int third = -(arr[i] + arr[j]);

        if (hashset.contains(third)) {
            List temp = Arrays.asList(arr[i], arr[j], third);
                temp.sort();
                st.add(temp);
        }
        hashset.add(arr[j]);
    }
}

return new ArrayList<>(st);
```

set to avoid duplicate

for finding if set is ordered otherwise O(1) in unordered

time - $O(n^2 \times \log M)$

space - $O(n) + O(\text{no of tuples}) \times 2$

when storing max element in hashset on first pass

arr = [-2, -2, -2, -1, -1, -1, 0, 0, 0, 2, 2, 2, 2]

Now we will first sort the array so that duplicate triplets can be avoided as same value would be together ∴ we can just move pointer till we find new element to consider

Now

Now (i) will start from beg. and
j from (i++) end
and K from (i+1)

we will add i+j+K element if sum>0 ∴ we need to decrease so the j-- will be done as array is sorted

this is how will find a triplet and once j crosses K no more triplets for that (i) and we will (i++)

[-2, -2, -2, -1, -1, -1, 0, 0, 0, 2, 2, 2, 2]
(i) (K) j

-2-2+2 = -2 <0 ∴ increase sum ∴ (++K)
as same element (-2) ∴ move

(-1)
Now -2-1+2 = -1 <0 ∴ K++

Now -2+2+0 = 0 triplet found

Now when triplet found K++, --j as both are useless increment / decrement till a new element is found.

arr = [-2 -2 -2 -1 -1 -1 0 0 0 2 2 2]

i ↑ j ↑ k ↓

crosses i ∴ no new element ∴ move(i)

arr = [-2 -2 -2 -1 -1 -1 0 0 0 2 2 2]

i ↑ j ↑ k ↑

$-1 + 0 + 2 = \underline{1}$ ∴ --k

arr = [-2 -2 -2 -1 -1 -1 0 0 0 2 2 2]

i ↑ j ↑ k ↑

$-1 + 0 + 2 = \underline{1}$ ∴ decrease(--k) it crosses

∴ move (i).

arr = [-2 -2 -2 -1 -1 -1 0 0 0 2 2 2]

i j k ↑

repeat.

```
List ans = ArrayList
Array.sort(arr);
for (i = 0 ⟶ n) {
        if ( i != 0 && arr[i] == arr[i-1]) continue;
        int j = i+1;
        int k = n-1;
```

to avoid duplicate by (i)

```
while (j<k){
    int sum = arr[i] + arr[k] + arr[j];
    if (sum < 0){
        j++
        }
    else if (sum > 0) {
        k--;
    }
    else {
        List temp = Array.asList(arr[i], arr[k], arr[j]);
        ans.add(temp);
        j++;
        k--;
        while (j<k && arr[j] == arr[j-1]) j++;
        while (j<k && arr[k] == arr[k+1]) k--;
    }
}
return ans;
}
```

removing duplicates

majedaar BC kuch bhi karte rehte hai ye log!!

sorting

time - $O(n^2) + O(n \log n)$

space → O(no of triplets)

4 SUM → problem statement is similar to 3 Sum problem just we have to get 4 elements two time

∴ OPTIMAL, BETTER, BRUTE would resemble 3 Sum

∴ starting only OPTIMAL

-2	-1	0	0	1	2
(i)	(j)	(k) →			← (l)

then instead of fixing (i) only we will fix (i) and (j) both and move (k) and (l) to find correct sum.

∴ after simple twist code

list ans;

for(i → n){ anoiding duplicate

if (i!=0 && arr[i] == arr(i-1)) continue;

for(j=i+1 → n){ anoiding duplicate

if(j=! i+1 && arr[j] == arr[j-1]) continue;

same code afterwards

just sum will contain 4 elements

}

}

time - $O(n^3)$

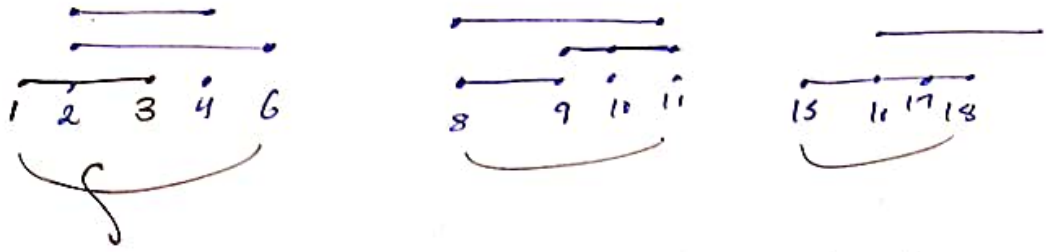
$n^2 \times n$ for inner while loop

for (j) and (i)

space - O(no of triplets) → the list which we are using to return answer.

MERGE OVERLAPPING SUBINTERVALS

- ((1,3) (2,6) (8,9) (9,11) (8,10) (2,4) (15,18), (16,17))



{ (1,6) , (8,11) , (15,18) } → output

BRUTE

(1,3) (2,6) (8,9) (9,11) (8,10) (2,4) (15,18) (16,17)

↓

sort (by first element if first is same then by second inter number)

↓

(1,3) (2,4) (2,6) (8,9) (8,10) (9,11) (15,18) (16,17)

as 6 < 8 not overlapping

(1, ~~3~~ ~~4~~ 6) → (1,6)

(1,3) (2,4) (2,6) (8,9) (8,10) (9,11) (15,18) (16,17)

already there

already there

new

(1,6)

(8,9) to 11 → (8,11)

similarly others

```
int n = arr.length;
Arrays.sort(arr);
List ans;
for (i=0 -> n) {
    start = arr[i][0]
    end = arr[i][1]
    if (!ans.isEmpty() && end <= ans.get(1)) continue;
    for (int j = i+1; j<n ; j++) {
        if (arr[j][0] <= end) {
            end = Math.max(end, arr[j][1]);
        }
        else break;
    }
    ans.add(Array.asList(start, end));
}
return ans;
}
```

included

get(size()-1) (1,4) (2,3)

check if overlaps (next one)

update answer array

time - $O(n \log n) + O(n+n)$

manually counted and figured it out approx as loop not running for all elements

for outer loop

space - $O(n)$

all distinct intervals.

MERGE TWO SORTED ARRAY WITHOUT EXTRA SPACE

arr[] = [1, 3, 5, 7] arr2[] = [0 2 6 8 9]

after ~~sorting~~ merging

arr = [0 1 2 3] arr2[] = [5 6 7 8 9]

OPTIMAL

arr1 = [1 3 5 7] arr2 = [0 2 6 8 9]

compare max with min now 7 > 0 ∴ swap

[1 3 5 ~~7~~ 0] [~~0~~ 2 6 8 9]

5 > 2 ∴ swap

[1 3 ~~5~~ 2 ~~7~~ 0] [~~0~~ ~~2~~ 5 6 8 9]

Now

[1 3 2 0] [7 5 6 8 9]

as 3 < 6 ∴

3 ← sorted 6 → sorted

no need comparing further

Now sort array individually

[0 1 2 3] [6 7 8 9]

arr1[] → n arr2[] – m;

```
int left = n-1;
int right = 0;
while ( left ≥ 0  && right < m ) {
        if (arr1[left] > arr2[right]) {
                swap(arr1[left], arr2[right]);
                left --;
                right ++;  }
        else break;
}
sort(arr1); sort(arr2);
}
```

TC → $O(\text{Min}(n, m)) + O(n \log n) + O(m \log m)$

as which ever smaller ~~bra~~ makes the condition of while false.

OPTIMAL (2) # GAP Method

[1 3 5 7] [0 2 6 8 9]

FIND REPEATING AND MISSING NUMBER

array = { 1, 3, 2, 5, 3 } $n = 5$

∴ number will be → 1, 2, 3, 4, 5

missing number (4)

repeated (3)

BRUTE

$O(n^2)$

```
for (i = 1 → n) {
    count = 0;
    for (j = 0 → n-1) {
        if (arr[j] == i) count + 1;
    }
    if (count == 2) repeating = i;
    else if (count == 0) missing = i;
    if (repeating != -1 && missing != -1) break;
}
```

BETTER

(use Hashing)

arr = [4, 3, 6, 2, 1, 1] $n = 6$

hasharray =

	1	2	3	4	5	6
	0	0	0	0	0	0
		2	1	1		1

(n+1)

Now iterate

2 → count = 2 (repeating)

5 → 0 = (missing)

```
int n = arr.size();
int hash[n+1] = {0};
for(i=0 ⟶ n){
        hash(a[i])++;
    }
int repeating = -1, missing = -1;
for(i ⟶ n){
        if (hash[i] == 2) repeating = i
        else if (hash[i] == 0) missing = i
        if (repeating != -1 && missing != -1) break
    }
}
```

✓ O(n)

✓ O(n)

time - O(2N)

space - O(N)

OPTIMAL (1)

arr = [4, 3, 6, 2, 1, 1] n = 6

Intuition - Sum = (4+3+6+2+1+1) = 17

sumN = (1+2+3+4+5+6) = 21

Now

$(\cancel{4}+\cancel{3}+\cancel{6}+\cancel{2}+1+\cancel{1}) - (\cancel{1}+\cancel{2}+\cancel{3}+\cancel{4}+5+\cancel{6}) = -4$

$1 - 5 = -4$

Repeating - missing = 4

$x - y = 4$ — (1)

Now we need another equation

$$(\cancel{4}^2+\cancel{3}^2+\cancel{6}^2+\cancel{2}^2+1^2+\cancel{4}^2)-(\cancel{4}^2+\cancel{2}^2+\cancel{3}^2+\cancel{4}^2+5^2+\cancel{6}^2)=24$$

$$1^2-5^2=24$$

$$x^2-y^2=24$$

$$(x+y)(x-y)=24$$

$$x-y \quad x+y=\frac{+24}{-4}$$

$$x+y=-6 \quad - \text{②}$$

Solve eq ① & eq ②

$\rightarrow$ $x=1$
$y=5$

```
long n= a.size();
// S-Sn ; -> x-y
// S2 -S2N;
SN = n(n+1)/2 , S, S2;
S2N = (n x (n+1) x (2xn+1))/6

for(i=0 -> n){
        S += a[i];
        S2 += (long long) a[i] * (long long) a[i];
}
long int val1 = S-Sn //(x-y)
int val2 = S2 - S2N //(x^2-y^2)
val2 = val2/val1 ; //(x+y)
```

```
long x = (val1 + val2) / 2      // 2x = sum

long y = x - val1;
return {(int) x, (int) y};
```

COUNT INVERSIONS

arr[] = {5, 3, 2, 4, 1}

count pairs where $a[i] > a[j]$ & $i<j$

Result → (5,3) (5,2) (5,4) (5,1)

→ (3,2) (3,1)

→ (2,1) = 8 pairs

→ (4,1)

BRUTE

- pick and count element to the right

count i

```
for (i → 0 → n-1) {
    for (j = i+1 → n) {
        if (a[i] > a[j]) count++;
    }
}
```

$O(n^2)$

OPTIMAL

Intution → 5 3 | 2 4 1

there we can form a pair from to array no matter where they are in their own array

if 3 5 | 1 2 4 → still can be used to make pairs from to array

sorted sorted

but if array are sorted pair making becomes effivient as

3 5 8 | 1 2 4

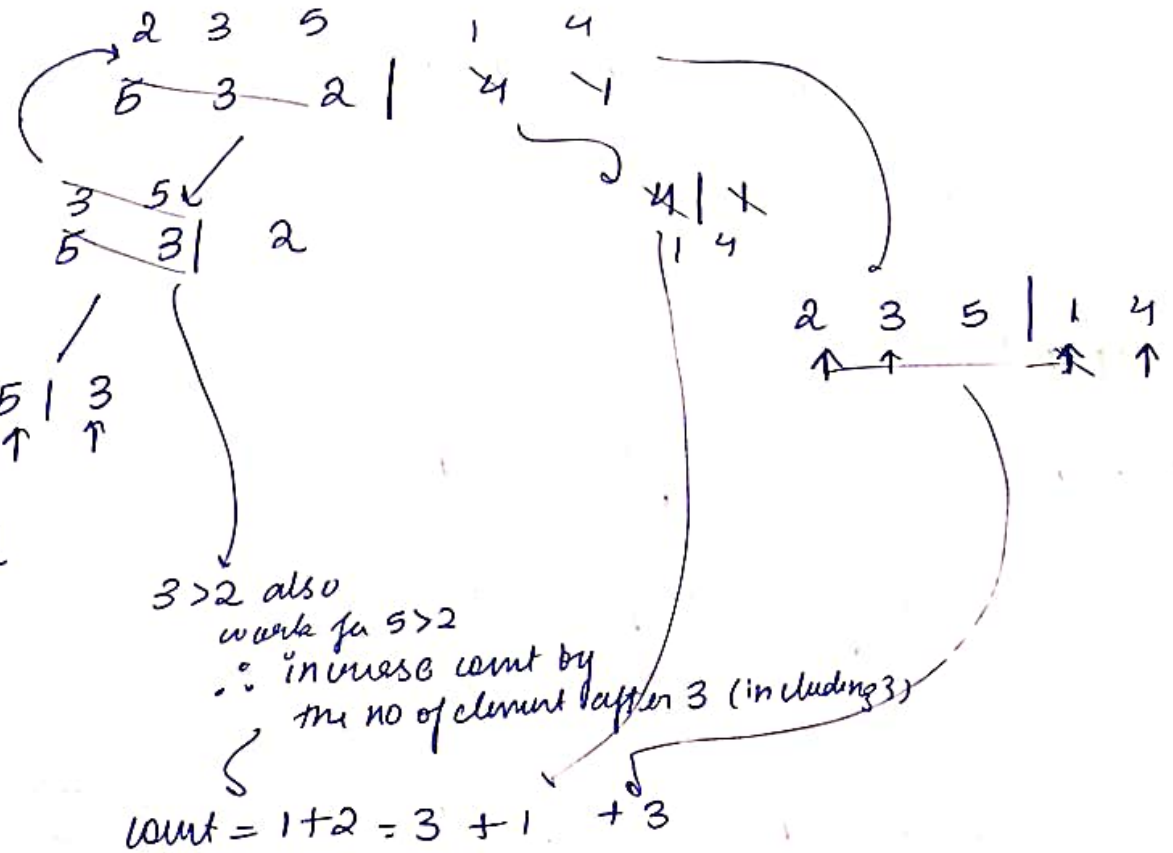
Now as 3>1 is true ∴ increase count but this will be true for all the other element after 3 also as they are sorted ∴ we can directly increase the count from 0 → to → 3

(3,1) (5,1) (8,1)

this leads to reduction in traversal ~~and comes down~~

∴ separating array both help and this happens
∴ using unsorted array in merge sort
∴ we will use merge sort to count.

code →



2 3 5 | 1 4 count = 4

sorted [1 mid - left + 1

true for all (2 > 1) length = ~~left~~

∴ count = 4 + 3 = 7

2 3 5 | 1 4 → sorted [1, 2

2 3 5 | 1 4

2 3 5 | 1 4 →

5 > 4 count++ ∴ sorted = [1, 2, 3, 4, 5]

$\boxed{\text{count} = 8}$

```
merge function ( ) {
  while ( left ≤ mid || right ≤ high ) {
    if ( arr[left] ≤ arr[right] ) {
        temp.push [arr[left]];
        left++;
    }
    else {
        temp.push (arr[right]);
        ~~sig~~ count = count + (mid - left + 1);
        right ++;
    }
}
```

this change only in merge sort code.

```
countInversion ( ) {
    mergeSort (arr, 0, ~~mid~~, high);
    return count;
}
```

as global.

Now global variable || are considered bad ∴
change code a bit

```
int merge (){
    count = 0;

    //same as on previous page
    return count;
}

int mergeSort (){
    if int count = 0;
      if (low ≥ high) return 0;
      mid = —
    count += mergeSort();
    count += mergeSort();
    count += merge;
    return count;
}
```

return individual count now

time - nlogn

space - O(n)

tell the interviewer that array will get modified ∴ if he say no modification on original data then create copy which will take extra space.

COUNT PAIRS

count no of pairs where

∴ i) $i < j$ &&

ii) $a[i] > 2 \times arr[j]$

→ ~~(40, 2) (40, 6)~~ .. sim

arr = [40, 25, 19, 12, 9, 6, 2]

total pairs = 15

OPTIMAL

sorted

if [6, 13, 21, 25] [1, 2, 3, 4, 4, 5, 9, 11, 13]

Now with count inversion approach

$6 = 3 \times 2$ (6, 3)

$6 = 6$ ∴ not a pair

but (13, 6) (21, 6) (25, 6)

are ∴ of our previous approach fails here as 8 4 as pointer will move to 4 and thus no pairs will be formed with 3.

∴ little different.

[6, 13, 21, 25] [1, 2, 3, 4, 4, 5, 9, 11, 13]

6 → [1, 2]

13 → [1, 2, 3, 4, 4, 5]

21 → [1, 2, 3, 4, 4, 5, 9]

25 → [1, 2, 3, 4, 4, 5, 9, 11, 13]

then if 6 forms a pair with (1 and 2) then all surrounding will form

∴ we can add for other pair which will form pair with 1, 2 in one go.

[6, 13, 21, 25] [1, 2, 3, 4, 4, 5, 9, 11, 13]

Now as (6,1) (6,2) count = 2

[6 13 21 25] [1 2 3 4 4 5 9 11 13]

count = 2 + 7

Now we do not start counting from 1 but from 3 as (1,2) will surely be a pair.

[6 13 21 25] [1 2 3 4 4 5 9 11 13]

all implicitly sure

count = 2 + 7 + 7

[6, 13, 21, 25] [1 2 3 4 4 5 9 11 13]

count = 2 + 7 + 7 + 8

this we will implement in merge sort

[6, 13, 21, 25] [1, 2, 3, 4, 4, 5, 9, 11, 13]

low mid mid+1 high

```
count = 0; right = mid+1;
for (i = low → mid) {
    while (right ≤ high && arr[i] > 2x arr[~~mid+1~~ right]) {
        right++;
    }
    count = count + (right - (mid+1));
}
return count;
```

// COUNT_PAIR FUNCTION

[6, 13, 21, 25] [1, 2, 3, 4, 4, 5, 9, 11, 13]

Now as (6,1) (6,2) count = 2

[6 13 21 25] [1 2 3 4 4 5 9 11 13]

count = 2 + 7

Now we do not start counting from 1 but from 3 as (1,2) will surely be a pair.

[6 13 21 25] [1 2 3 4 4 5 9 11 13]

all implicitly sure

count = 2 + 7 + 7

[6, 13, 21, 25] [1 2 3 4 4 5 9 11 13]

count = 2 + 7 + 7 + 8

thus we will implement in merge sort

[6, 13, 21, 25] [1, 2, 3, 4, 4, 5, 9, 11, 13]

low mid mid + 1 high

```
count = 0; right = mid + 1;
for (i = low → mid ){
    while ( right ≤ high && arr[i] > 2x arr[~~mid+1~~ right] ){
        right ++;
    }
    count = count + (right - (mid + 1));
}
return count;
```

|| COUNT_PAIR FUNCTION

code →

```
int mergeSort ){
    int count = 0;
    if ( low ≥ high) return count;

count += mergSort ( );
count += mergeSort ( );
    count += countPairs (arr, low, mid , high);
    merg ( );
return count ;
}
```

time → $\log n \times (n + n)$

divisions — merge — as every element is visited from both divisions multiple visits are not there.

space - $O(n)$ — but modifying array minimum that.

MAXIMUM ARRAY PRODUCT
subarray

- arr

arr = [2, 3, -2, 4]

6 max sub array product

arr = [-2, 0, -1]

0 max subarray product

BRUTE

make all possible sub arrays

[2 3 -2 4]

```
for (i=0 ⟶ n){

    for (j=i ; j<n ; j++){
        prod = 1
        for (k=i ⟶ j) {
            product = product × arr[k];
        }
        max = Math.max(max, product);
    }
}
```

time - ≈ $O(n^3)$

BETTER

```
max = INT_MIN;
for (i = 0 → n) {
    product = 1
    for (j = i → n) {
        product = product × arr[j];
        max = max(max, product);
    }
}
```

$O(n^2)$

OPTMAL

OBSERVATION →

1) → all +ve
2) → even negative, (+ve rest)
] multiply everyone

3) → odd negative, (other +ve)

[3 2 (-1) 4 (-6) 3 (-2) 6]

if

- [$\overbrace{\boxed{3 \quad 2}}^{\text{prefix}}$ ✗ $\overbrace{\boxed{4 \quad -6 \quad 3 \quad -2 \quad 6}}^{\text{suffix}}$]

ignore

either of both

- [$\underbrace{3 \quad 2 \quad -1 \quad 4 \quad -6 \quad 3}_{\text{prefix}}$ ~~-2~~ $\underbrace{6}_{\text{suffix}}$]

[$\underbrace{3 \quad 2 \quad -1 \quad 4}$ ~~-2~~ $\underbrace{3 \quad -2 \quad 6}$]

↳ Not good case compare to above cases

1) if it has zero.

$\boxed{-2 \quad 3 \quad 4 \quad -1}$ 0 $\boxed{-2 \quad 3 \quad 1 \quad 4}$ 0 $\boxed{4 \quad 6 \quad -1 \quad 4}$

mom mom mom

Now dry run

[2 3 -2 4]

prefix sum = 2×3 = 6 × -2 = -12 × 4 = -48

mom = Intline 2 & 6

postfix sum = 4 ≠ 4×-2 = -8 × 3 × 2 = 48

approach is we calculate from both side the mom is either in prefix or suffix will automatically gets updated.

if -2 3 4 -1 0 -2 3 1 4 0 4 6 -1 4

if (prefix == 0) prefix = 1

similarly for ~~postfix~~ suffix.

```
pre = 1  suff = 1
for (i = 0 ——→ n) {

    if (pre == 0) pre = 1;
    if (suff == 0) suff = 1;

    pre = pre x arr[i];
    suff = suff x arr[n-i-1];
moni = ~~mom(pre, suff)~~ Max(mom, mom(pre, suff))
}
```

O(n)

ARRAY SEARCH IN ROTATED DUPLICATE ELEMENTS

arr[] = [3 1 2 3 3 3 3] ← rotated

low, mid, high

all are same ∴ no distinction between sorted and unsorted half.

∴ previous fails...

answer → from down this condition

arr = [3 1 2 3 3 3 3]

[3 (1 2 3 3 3) 3]

∴ shrink space until

arr[mid] == arr[low] == [high]

not true

```
if (arr[mid] == target) return mid;
```

after this

```
if (arr[low] == arr[mid] && arr[mid] == [high]) {
        low++;
        high--;
        continue;
}
```

after this same as previous.

time → $O(\log_2 n)$ → average

but [3 (3 (1 3 3) 3) 3]

$\simeq O(n/2)$

where lot of duplicates.

SEARCH IN ROTATED SORTED ARRAY

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]

[7, 8, 9, 1, 2, 3, 4, 5, 6] rotated sorted array

Now search for (n)

7 8 9 1 (2) 3 4 5 6

Now one part will always be sorted either right or left of mid

here right of mid

now $\boxed{n = 8}$

check if $2 \le 8$ ✓ but $8 \le 6$ ✗

∴ not present in right half

∴ we eliminate half by comparing element with sorted part

0 1 2 3

(7) (8) 9 (1) 2 3 4 5 6

sorted unsorted

now $7 \le 8$, $9 \le 8$, $8 \le 8$ here mid == 8 ∴ break

∴ eliminate unsorted part

7 == 8