Problems in Sorting & Hashing.

Problems in Sorting.

(1) Merge overlapping intervals.

I/p → {{1,3}, {2,4}, {5,7}, {6,8}}
O/p → {{1,4}, {5,8}}

} Need not be sorted.

Sol. For this question we have to use pairs in java

(OR)

we can define our own class.

⇒
```
Class Interval {
    int start;
    int end;
}

Interval [] arr = new Interval[n];
```
} (1)

Step 1
Check if two intervals overlap.

i) Take larger value of start b/n 2 Intervals.
Does that larger value lie b/n interval of other pair?

[5, 10]
[1, 7]
} 5 lies in-b/n ∴ overlap.

ii) You can also consider smaller end value

Naive sol

- $O(n^3)$ → ($O(n^2)$ 2 loops * $O(n)$ deleting)
- 2 loops
- Take one interval and check with all other intervals, if it overlaps.
- b/n 2 intervals.
  - {5,10}, {2,7}
    - start = min(i1.start, i2.start);
      end = max(i1.end, i2.end);
      } After checking overlapping.

Efficient sol.

- $O(n \log n)$ time

1) Sort the intervals by start time.

2) Here we don't compare pairs, we compare a pair with prev merged intervals so far.

Code

After (1)

```
Void mergeIntervals(Interval arr[], int n)
{ sort(arr, arr+n, MyComp);   → For sorting our class, MyComp interface has to be used.
  int res = 0;
  for(int i=1; i<n; i++)
  { if(arr[res].end >= arr[i].st)
    {
      arr[res].end = max(arr[res].end, arr[i].end);
      arr[res].st = min(arr[res].st, arr[i].st);
    }
```

```
        else { res++; arr[res] = arr[i]; }
    }
    for(int i=0; i<=res; i++)
        { print(arr[i].st + " " + arr[i].end); }
}
```

---

(2) Meeting Maximum Guests

800 → 8:00

Q.

I/P : arr {800, 700, 600, 500}

: dep {840, 820, 830, 530}

O/P : 3.

Arrival & depa times of guests are given, you have to meet max guest.

O/P may be asked → Optimal time to go. (600)

Max no. of guest you can meet.

Sol

Variation of find overlapping intervals.

Q = {900, 600, 700
{1000, 800, 930

⇒ Explanation

Time	A/Dept	No: of guests at party (curr)
600	A	1
700	A	2
730	D	1
800	D	0
900	A	1
1000	D	0

So max guests we can meet is 2.

Algo. (2 pointer used).

```
int maxGuest (int arr[], int dep[], int n)
{
    sort(arr); ⟶ This is also Arrays.sort(arr);
    Arrays.sort(dep);
    int i=1, j=0, res=1, curr=1;
    while(i<n && j<n)
    { if(arr[i]<=dep[j])
        { curr++; i++; }
      else { curr--; j++; }
      res = max(res, curr);
    }
    return res;
}
```

(3) Smallest kth element.

* I/p: {10, 5, 30, 12}
  k = 2

O/p : 10.

Sol M-1
Sort the elements and return arr[k-1]; } Naive.

```
int KthSmallest (int arr[], int n; int k)
{
    Arrays.sort(arr);
    return arr[k-1];
}
```

O(nlogn);

If modifications not allowed then you can copy to another array and sort → O(n) space added.

M-2

Modification and O(1) space. → Quick Select Algorithm

:- → Lomuto partition. → places pivot at correct position.
(O(n2) - worst but arg is better) if (k-1 == p) return.

```
int KthSmallest (int arr[], int n, int k)
{
    int l=0, r=n-1;
    while (l<=r)
    {
        int p = partition (arr, l, r);
        if (p==k-1) {return p;}
        else if (p>k-1) { r = p-1; }
        else { l = p+1; }
    }
}
```

(4) Minimum Difference in an Array.

» I/p: arr[] = { 1, 8, 12, 5, 18 }

O/p: 3

I/p: arr = { 8, -1, 0, 3 }

O/p: 1

Sol.

» Naive sol

↳ O(n2) » Remember to find abs diff.

» traverse diff for each ele to every other element & update min(curr, diff);

Effecient sol.

» Sorting and then finding diff will take only one loop.

O(nlogn).

Algo.

(1)
```
for(int i=1 ; i<n ; i++){
    for(int j=0; j<i ; j++) {
        res = min(res, abs(arr[i]-arr[j]));
    }
}
```

(2) Only inner loop after sorting.

(5) Chocolate Distribution Problem.

I/p : {7, 3, 2, 4, 9, 12, 56} → each ele in a packet & contains no: of chocl i.e equal to it's value.

m → no: of children = 3.

Rules

1) Each child gets only one packet.

2) Min diff b/n (child getting min chocletes) & (child getting max chocletes)

O/p: 2.

Explan → 3, 2, 4 are picked, min diff → 2. acc to 2) rule

Sol.

Sort and use sliding window..

∴ {2, 3, 4, 7, 9, 12, 56}

4 − 2 → (arr[i+m−1] − arr[i])

→ int minDiff(int arr[], int n, int m)

```
{
    if (m>n) return -1;
    Arrays.sort(arr);
    int res = arr[m-1] - arr[0];
    for (int i=1; (i-m-1)<n; i++)
    {
        res = min (res, (arr[i+m-1] - arr[i]));
    }
}
```

(6) Sort an array with 2 types of elements.

3 forms.

1) Segregate positive and negative.

I/p: {15, -3, -2, 18}

O/p: {-3, -2, 15, 18}

2) Segregate even and odd.

I/p: {15, 14, 13, 12}

O/p: {14, 12, 15, 13}

3) Sort a Binary array.

I/p: {0, 1, 1, 1, 0}

O/p: {0, 0, 1, 1, 1}

Sol.

Naive (my sol)

Start one type from start
Start other type from end
store in new array temp.

} 2 pointers approach
O(n) & O(n)

Naive.

→ Take a new array temp.
put all -ve in start of temp.
copy rest of +ve numbers.

} space O(n)
O(n) time

Efficient sol is to apply partition.

```
void segPosNeg ( int arr[], int n)
{
    int i=-1, j=n;
    while (true)
    {
        do { i++; } while (arr[i]<0);
        do { j--; } while (arr[j]>=0);
        if (i>=j)
            return;
        swap (arr[i], arr[j]);
    }
}
```

⑤ Count Inversions in an array.

I/p: {2, 4, 1, 3, 5}

O/p: 3

Exp → {{4, 3}, {4, 1}, {2, 1}}

Inversion
↳ A pair (arr[i], arr[j]) forms an inversion when arr[i] > arr[j] & i < j.

* A sorted array has zero inversions.

⇒ Sol

Naive

⇒ 2 loops

```
for (i=0 - - ) {
    for (j = i+1 - - ) {
        if (arr[i] > arr[j])
            res++;
    }
}
```

⇒ Eff

count inversions while doing merge sort.

⇒ Pls refer merge fn discussed before.