1. Merge Intervals
   We will sort our intervals and merge intervals one by one. TC is O(nlogn)

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
      if not intervals:
        return []
      intervals.sort()
      result = [intervals[0]]
      for interval in intervals[1:]:
        if not result[-1][1] < interval[0]:
          result[-1][1] = max(interval[1], result[-1][1])
        else:
          result.append(interval)
      return result
```

2. Meeting Rooms
With the same method, we will sort our intervals and compare adjacent intervals, once there is overlap, we will return False, or in the end, we will return True. TC is O(nlogn)

```
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        if len(intervals) < 2:
          return True
        intervals.sort()
        cur = intervals[0]
        for interval in intervals[1:]:
          if cur[1] > interval[0]:
            return False
          cur = interval
        return True
```

3. Meeting Rooms II
We will sort all intervals and append each ending time to heapq, each time we will top of heapq to interval[0] if heapq[0] <= interval[0], we could use this room and pop out this ending time. TC is O(nlogn)

```
from heapq import *
class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
      intervals.sort()
      hq = []
      for interval in intervals:
        if hq and hq[0] <= interval[0]:
          heappop(hq)
        heappush(hq, interval[1])
    return len(hq)
```

4. Top K Frequent Elements
We will use Counter to count every element's presenting times. Then use a heap queue to push (num, count) to hq, if its length is larger than k, we will pop the smallest one. TC is O(nlogk)

```
from heapq import *
from collections import Counter
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        memo = Counter(nums)
        hq = []
        for num, count in memo.items():
          if len(hq) >= k:
            heappushpop(hq, (count, num))
          else:
            heappush(hq, (count, num))
        return list(map(lambda a: a[1], hq))
```

5. Top K Frequent Elements
We will use Word to encap our key, value and custom our comparator. The other things would be like Top K frequent elements, but we need to sort it in the end. TC is O(nlogk)

```
from collections import Counter
from heapq import *
class Word:
    def __init__(self, freq, word):
        self.freq = freq
        self.word = word

    def __lt__(self, other):
        if self.freq != other.freq:
            return self.freq.__lt__(other.freq)
        else:
            return self.word.__gt__(other.word)


class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        memo = Counter(words)
        h = []
        for key, value in memo.items():
          heappush(h, Word(value, key))
          if len(h) > k:
            heappop(h)
```

```
h.sort(reverse=True)
return list(map(lambda x: x.word, h))
```