1. Find the index of the smallest number in a sorted array, where the first k numbers were shifted to the end.

1. How would you find the index of the smallest number.

   A brute-force solution is to ignore the sorted array property and scan every element in the array to find the smallest element. This is $\Theta(n)$.

   An algorithm that uses the divide-and-conquer principle is to first start from the leftmost index and the rightmost index as our list of candidates. Compare the middle element of that interval with the $A[right]$. If $A[right] < A[mid]$ then since $A[right]$ is the largest element in that subarray we cut the candidate size fo $[mid+1, right]$. else $A[mid] < A[right]$ then that subarray could not contain the smallest element and we reduce the size to candidate size to $[left, mid]$.

2. Write the pseudocode for the best algorithm you came up with.

Algorithm 1 min_element_in_rotated_array

```
prodedure(A) :
left := 0, right := len(A) − 1
while left ≠ right :
    mid := ⌊(left+right)/2⌋
    if A[mid] < A[right] :
            right := mid
    else :
            left := mid + 1
return left
```

3. Implement your answer using any programming language you want.

```cpp
#include <iostream>
#include <vector>

using namespace std;

/**
 * T(n) = T(n / 2) + O(1)
 * T(n) = O(log_2(n))
 */
int min_element_in_rotated_array(const vector<int>& A){
    int left = 0, right = A.size() - 1;
    while(left < right){
        int mid = (left + right) / 2;
        if(A[mid] > A[right]){
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return left;
```

```
21 }
```

4. What is the time complexity of your answer?

   The recurrence is of the form $T(n) = T(\frac{n}{2}) + O(1)$ which evaluates to $T(n) = O(log_2(n))$.

2. You are given k sorted arrays in descending order of size n. Develop an algorithm to merge them into a single sorted array of size $k \cdot n$.

1. How would you merge these k arrays into a single sorted array?

   I would allocate an array of size $n \cdot k$. Then for each array I would store the element, the index of that array and the index of the curent element in that array into a maxheap that does comparisons based on the element. we continously extract the max element place it into the allocated array and then increment the index of the array the element belonged to.

2. Write the pseduocode for the best algorithm you came up with.

Algorithm 2 Merging k arrays

```
1: prodedure(A) :
2: maxPQ:= MAXHEAP()
3: result = ARRAY()
4:     for(int i = 0 ;i < A.size() ;i++) :
5:             maxPQ.insert(Node{A[i][0], i, 0})
6:      while maxPQ.size() ≠ 0 :
7:              Node temp:= maxPQ.extractMax()
8:              result.insert(temp.element)
9:              if(temp.current_index<A[temp.array_index) :
10:                new_index:=temp.current_index+1
11:                array_index := temp.array_index
12:               maxPQ.insert({A[array_index][new_index],array_index,new_index)
13: return result
```

3. Implement your answer using any programming language you want.

```
1  #include <iostream>
2  #include <queue>
3  #include <vector>
4
5  using namespace std;
6
7  struct Node{
8      int element;
9      int array_idx;
10     int curr_idx;
11
```

```
12      bool operator>(const Node& rhs) const{
13          return element > rhs.element;
14      }
15
16      bool operator<(const Node& rhs) const{
17          return element < rhs.element;
18      }
19  };
20
21  vector<int> merge_k_array(const vector<vector<int>>& A){
22      vector<int> result;
23      priority_queue<Node> maxPQ;
24      for(int i = 0;i < A.size();i++){
25          maxPQ.push(Node{A[i][0], i, 0});
26      }
27      while(!maxPQ.empty()){
28          Node temp = maxPQ.top();
29          maxPQ.pop();
30          result.emplace_back(temp.element);
31          int new_idx = temp.curr_idx + 1;
32          int array_idx = temp.array_idx;
33          if(new_idx < A[array_idx].size()){
34              maxPQ.push(Node{A[array_idx][new_idx], array_idx,
                    new_idx});
35          }
36      }
37      return result;
38  }
```

4. What is the time complexity of your code?

   The time complexity of this algorithm is $O(n \cdot k \cdot log_2(k))$. There are in total $n \cdot k$ total elements to examine and the heap holds at most k elements. any insertion and extraction of the heap takes $log_2(k)$, hence the total time complexity is $O(n \cdot k \cdot log_2(k))$.