Assignement 1

Question 1

```
Pseudo code
Note : idexing is from 1 to N for the below case
We start by looping over the array.

1. A= [31,41,59,26,41,58]
   i=1, A[1]=31; Since this is the first element  we would only start checking from the
2nd postion and skip this one


2. A= [31,41,59,26,41,58]
    i=2, A[2]=41; we will check if this element is less than the previous element i.e
A[1] = 31. As 41 is greater than 31, it is in the correct order and we would move to the
next element.


3. A = [31,41,59,26,41,58]
    i=3, A[3]=59; Since 59 is greater than the previous element i.e A[2] = 41 , we will
move to the next index

4. A= [31,41,59,26,41,58]
    i=4, A[4]=26 which is less than 59 we'll move the element 59 to position 4 and 26 to
position 3
    now A =  [31,41,26,59,41,58]

    we observe again that, A[3] = 26 is less than previous element 41
    hence we move the element 41 to position 3 and element 26 position 2
    now A = [31,26,41,59,41,58]

    we observe again that, A[2] = 26 is less than previous element 31
    hence we move the element 26 to position 1 and element 31 position 2
    now A =  [26,31,41,59,41,58]


5. i=5, A[5] = 41; Since 41 < A[4] = 59 we will interchange these elements.
    A = [26,31,41,41,59,58]




6. i=6, A[6]= 58 ; As 58 < 59 and we intercgange these elements
    A = [26,31,41,41,58,59]

We have hence got a sorted array
```

In [ ]:

```
##Question 1
def insertionSort(ListOfNum):

    totalNums = len(ListOfNum)

    for i in range(1, totalNums):

        numberToCheck = ListOfNum[i]

        j = i-1

        while(j>=0 and ListOfNum[j]>numberToCheck):
            ## move all the elements to one step right
            ListOfNum[j+1] = ListOfNum[j]
            j  = j - 1
        ###at the required position insert the element
        ListOfNum[j+1] = numberToCheck

    return True
```

Question 2

```
def unknown(ListOfNum):                              cost   times
    totalNums = len(ListOfNum)                       c1      1
    for i in range(totalNums-1):                     c2      n-1 times
        if ListOfNum[totalNums-1] < ListOfNum[i]:    c3      n-1 times
            temp = ListOfNum[i]                      c4      n-1 times
            ListOfNum[i] = ListOfNum[totalNums-1]    c5      n-1 times
            ListOfNum[totalNums-1] = temp            c6      n-1 times


    return ListOfNum[totalNums-1]
```

In [ ]:

```
##Question 2
def unknown(ListOfNum):
    totalNums = len(ListOfNum)
    for i in range(totalNums-1):
        if ListOfNum[totalNums-1] < ListOfNum[i]:
            temp = ListOfNum[i]
            ListOfNum[i] = ListOfNum[totalNums-1]
            ListOfNum[totalNums-1] = temp


    return ListOfNum[totalNums-1]
```

1. As we loop across the array since at every step we wouold be comparing the current element at last index with the loop index and replacing the largest of the two at the last index , at the end of the loop we would have the maximum array at the position.

2. The running time of the algorithm is same as finding the maximum i.e
   $c_1 + (c_2 + c_3 + c_4 + c_5 + c_6) * (n - 1)$ = O(N-1) = O(N)

Question 3

Basically pick up every list intersect with the next list in the same way we do merging , just that when the elements are equal keep dumping them in another list. Now after completing N+N elemets of the first and second list , re assign the common elemetns to the old intersection list and continue this process till all lists are collapsed. This takes 2N for comparing two lists and getting the common elements and since there are K-1 list pairs i.e 1,2 and 2,3 and 3,4 so on.... we have the complexity as
O(2(k-1)N) = O(kN)

```
## pseudo code

intersectionElements = inputListOfLists[0] // assign first list as the intersection list

for eachList in inputListOfLists[1:]: // pick up each array in the list of the lists and
progressively intersect
                                      // with the next list

    LenI = len(intersectionElements)
    LenJ = len(eachList)

    while(indexIntersection < LenI and indexNextList < LenJ):


        if intersectionElements[indexIntersection] == eachList[indexNextList]: // if the
elements in the previous result of
                               // intercsection are equal , move both indexes

            newIntersectionList.append(intersectionElements[indexIntersection])
            indexIntersection += 1
            indexNextList += 1


        elif intersectionElements[indexIntersection] > nextList[indexNextList]: //else
move only the index that is lesser value
            indexNextList += 1

        elif intersectionElements[indexIntersection] < nextList[indexNextList]:
            indexIntersection +=1
    intersectionElements = newIntersectionList

return intersectionElements
```

In [2]:

```
##question 3

## O(2(k-1)N) solution
def findCommonElements(inputListOfLists):
#    inputListOfLists = [[1,3,5,7,9,11],[2,4,6,8,9,11],[4,5,6,7,9,11]]
#    k = 3
#    N = len(inputListOfLists[0])
    oldIntersectionList = inputListOfLists[0]

    for nextList in inputListOfLists[1:] :
        newIntersectionList = []
        indexIntersection = 0
        indexNextList = 0
        LenI = len(oldIntersectionList)
        LenJ = len(nextList)
        while(indexIntersection < LenI and indexNextList < LenJ):
            if oldIntersectionList[indexIntersection] == nextList[indexNextList]:
                newIntersectionList.append(oldIntersectionList[indexIntersection])
                indexIntersection += 1
                indexNextList += 1


            elif oldIntersectionList[indexIntersection] > nextList[indexNextList]:
                indexNextList += 1

            elif oldIntersectionList[indexIntersection] < nextList[indexNextList]:
                indexIntersection +=1
        oldIntersectionList = list(newIntersectionList)

    return oldIntersectionList
```

In [15]:

```
## a exact O(KN) solution using dictionaries
def findCommonElements(inputListOfLists):
    commonElem = {}
    k = len(inputListOfLists)

    for i in range(len(inputListOfLists[0])):
        for j in range(k):

            elem = inputListOfLists[j][i]

            if elem not in commonElem:
                commonElem[elem] = 1
            else:
                commonElem[elem]+=1
                if commonElem[elem]== k:
                    print(elem)
```

Question 4

```
Pseudo code
```

```
add 2 extra spaces to the List at the end and denote them by -1

inputList = inputList + [-1,-1]  // [2,3,4] = [2,3,4,-1,-1]


for index, corresponding value in inpList:
    // keep putting the value at their right postions and break if a empty slot comes up
    while inpList[value] != value and value != -1:
         interchange inpList[value] and inpList[index]


##output all the missing numbers wherever there is -1 which denotes an empty slot

for i,value in enumerate(inpList):
    if value == -1: // missing number as there is an empty slot
        print(i)
```

Though there is a while inside for, the total number of iterations that happen is N , as no element that is in its correct postions is reshuffled again. Hence the while loop is executed only when there is element misplaced at the ith position

In [8]:

```
##question 4
def find2MissingNums(inpList,numMissing):
    ##trails

    inpList.extend([-1 for i in range(numMissing)]) ##denoting empty spaces by -1


    for position,value in enumerate(inpList):
        while inpList[value] != value and value != -1:
            temp1 = inpList[value]

            inpList[value] = value
            inpList[position] = temp1
            value = inpList[position] #updated value
    ##print all the missing numbers
    for i,value in enumerate(inpList):
        if value == -1: ##missing
            print(i)
```

Question 5

a) below are the five inversion pairs for the given array by definition [2, 3, 8, 6, 1] :
With elements,
i. (1,2) ii. (1,3) iii. (1,8) iv. (1,6) v. (6,8)
at Index postions
i. (1,5) ii. (2,5) iii. (3,4) iv. (3,5) v. (4,5) respectively

b) An array sorted in descending order would have the most inversions since every i^{th} element would have i-1 inversions i.e
Sum of all inversions are, 1+2+..n-1 which is $(n-1)(n)/2$

c) The running time or number of iterations is directly proportional to the number of inversions in the array.Infact the running time \cost is constant times the number of inversions. As at each iteration (while loop) we would check the indices previous to that,and would only break if there are no more inversions.

Question 6

```
## Took help from
## https://stackoverflow.com/questions/4607945/how-to-find-the-kth-smallest-element-in-
the-union-of-two-sorted-arrays

#### pseudo code

def getKthLargestElem(k,listM,listN):


    if one of the array becomes empty
        retturn the kth element of the non empty array.

    // capture the first k elements of the m , n arrays
    // since the kth largest cannot exist beyond the kth elements.

    listM = listM[:k]
    listN = listN[:k]

    newListM = list(listM)
    newListN = list(listN)

    midElemIndexM = int(len(newListM)/2) // get the mid element of the both arrays
    midElemIndexN = int(len(newListN)/2)

    If k <= midElemIndexM + midElemIndexN :  // if kth value lies inside the mid
elemenths of the both arrays
        if newListM[midElemIndexM] < newListN[midElemIndexN]:

            // neglect values that are greater than mid Nth elem in N array

            kthElem =  getKthLargestElem(k,newListM,newListN[:midElemIndexN])

        else if (newListM[midElemIndexM] > newListN[midElemIndexN]):

            // neglect values that are greater than mid Mth elem in M array

            kthElem =  getKthLargestElem(k,newListM[:midElemIndexM],newListN)

    If k > midElemIndexM + midElemIndexN:



        if newListM[midElemIndexM] < newListN[midElemIndexN]:

            // neglect values that are lesser than mid Mth elem in M array
            // adjust K such that the lesser numbers are already in your pocket and
owuld only wnat the k-pocketed numbers
```

```
            kthElem =  getKthLargestElem(k-midElemIndexM-
1,newListM[midElemIndexM+1:],newListN)

        else if (newListM[midElemIndexM] > newListN[midElemIndexN]):

            // neglect values that are lesser than mid Nth elem in the N array
            // adjust K such that the lesser numbers are already in your pocket and
would only want the k-pocketed numbers

            kthElem =  getKthLargestElem(k-midElemIndexN-
1,newListM,newListN[midElemIndexN+1:])




    return kthElem
```

Question 7

1. Generate a random number in the range of list indeices call it parentIndex as this would be the index of the element that is being transferred from the parent array
2. Then generate another random integer within the range of the indices call it empty index denoting the index position of the
3. Transfer the element at parent index at empty index place if the element is empty at the empty array. Else generate another random_int
4. This way at every iteration each element has equal chance to be chosen to got at any index in the empty array.

```
## Pseudo code

def permuteNumbers(listOfNums):
    lengthN = len(listOfNums)
    totalTransferredIndices = 0 // total indexes transferred to another array

    while totalTransferredIndices < lengthN :
        randomParentIndex = random_num_in_index_range
        if randomParentIndex not in allIndexes:
            allIndexes.append(randomParentIndex) // keep track of all idexes
            randomEmptyIndex = random_num_in_index_range
            if emptyArray[randomEmptyIndex] is empty:
                emptyArray[randomEmptyIndex] = listOfNums[randomParentIndex]
                totalTransferredIndices+=1
```

In [13]:

```
###this code takes large amount of time
import random
def permuteNumbers(listOfNums):
    lengthN = len(listOfNums)
    totalTransferredIndices = 0
    allIndexes = []
    emptyArray = ['NOT_INT' for i in range(lengthN)]
    while totalTransferredIndices < lengthN :
        randomParentIndex = random.randint(0,lengthN-1)
        if randomParentIndex not in allIndexes:
            allIndexes.append(randomParentIndex)
            randomEmptyIndex = random.randint(0,lengthN-1)
            if emptyArray[randomEmptyIndex] == 'NOT_INT':
                emptyArray[randomEmptyIndex] = listOfNums[randomParentIndex]
                totalTransferredIndices+=1
```

In [14]:

```
## another version of ideas borrowed from fisher-yates shuffle alogorithm
## https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

def permuteNumbers(listOfNums):
    lengthN  = len(listOfNums)
    for i in range(len(listOfNums)):

        j =  random.randint(0,lengthN-1)
        temp = listOfNums[i]
        listOfNums[i] = listOfNums[j]
        listOfNums[j] = temp
    return listOfNums
```

Out[14]:

```
[3, 4, 2]
```

Above code geenrates a random interger and since and swaps it with another index that way each of the array element has opportunity to be present at each of the index , hence equal probability.