COMP 352

Tutorial Session 10

OUTLINE

- Sorting properties
- Sorting algorithms
  - Quicksort
  - Mergesort
  - Bucket sort
  - Radix sort
- Exercise

- For live demo please check

https://www.toptal.com/developers/sorting-algorithms

Sort Algorithm Properties

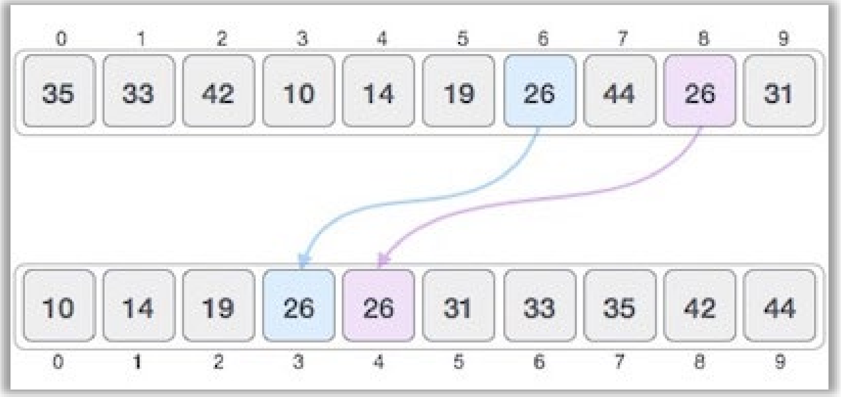
In-place Sorting and Not-in-place Sorting

- Sorting algorithms may require extra space for comparison and temporary storage of data elements
- A sorting algorithm is ***in-place*** if
  - it uses no auxiliary data structures (however, O(1) auxiliary variables are allowed)
  - it updates the input sequence only by means of operations replaceElement and swapElements
- Bubble sort is an example of in-place sorting and Merge-sort is an example of not-in-place sorting.

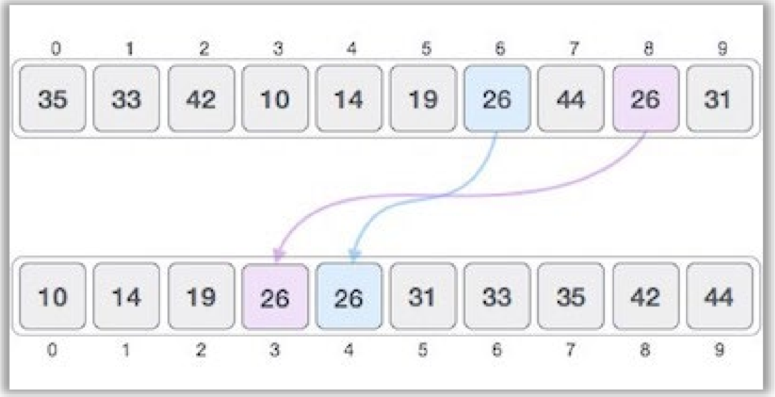
Sort Algorithm Properties

Stable Sort

- If a sorting algorithm, after sorting the contents, does not change the sequence of appearance of duplicates relative to the original ordering, it is called **stable sorting**.



- If a sorting algorithm, after sorting the contents, changes the sequence of appearance of duplicates relative to the original ordering, it is called **unstable sorting**.



- Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

QUICKSORT: OUTLINE

Recursive method:

Input: array, firstindex, lastindex

1. Check the stopping case: firstindex<lastindex
    1. Find the splitpoint : partition→Most important point NEXT SLIDE!!!!
    2. Recursion on left part
    3. Recursion on Right part

The Quicksort : algorithm

Partition: return the pivot position

1) Choose a pivot

2) Set a left pointer and right pointer

3) Compare the left pointer element (lelement) with the pivot and the right pointer element (relement) with the pivot.

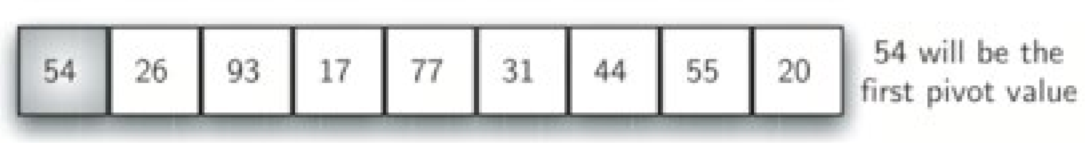
4) Check if lelement<pivot and relement>pivot:

   a. If yes, increment the left pointer and decrement the right pointer
   b. If not, swap the lelement and relement

5) When left >= right, swap the pivot with either left or right pointer.

PARTITION ALGORITHM: EXAMPLE

1. Choosing the pivot:

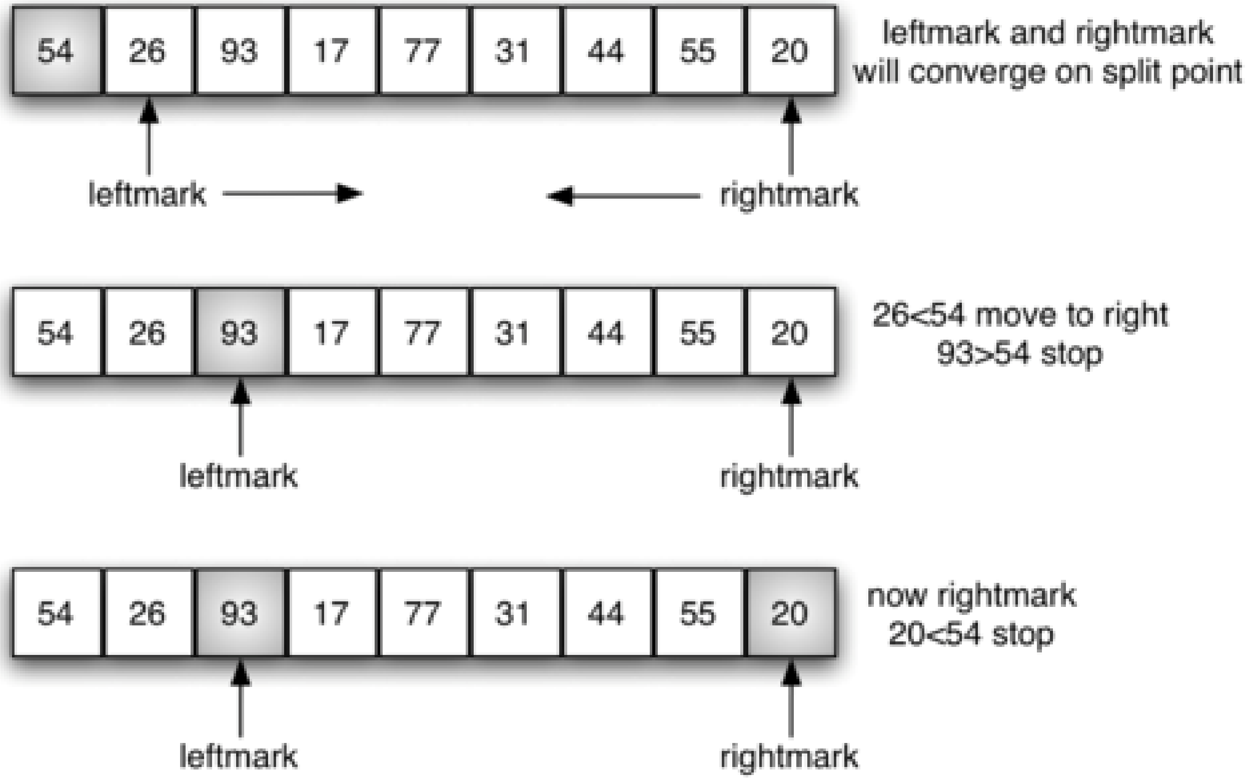


54	26	93	17	77	31	44	55	20

54 will be the first pivot value

2. Moving through the array to find the last position of the pivot: the partition

PARTITION: CONT'D



54	26	93	17	77	31	44	55	20

leftmark and rightmark will converge on split point

leftmark → ← rightmark

54	26	93	17	77	31	44	55	20

26<54 move to right
93>54 stop

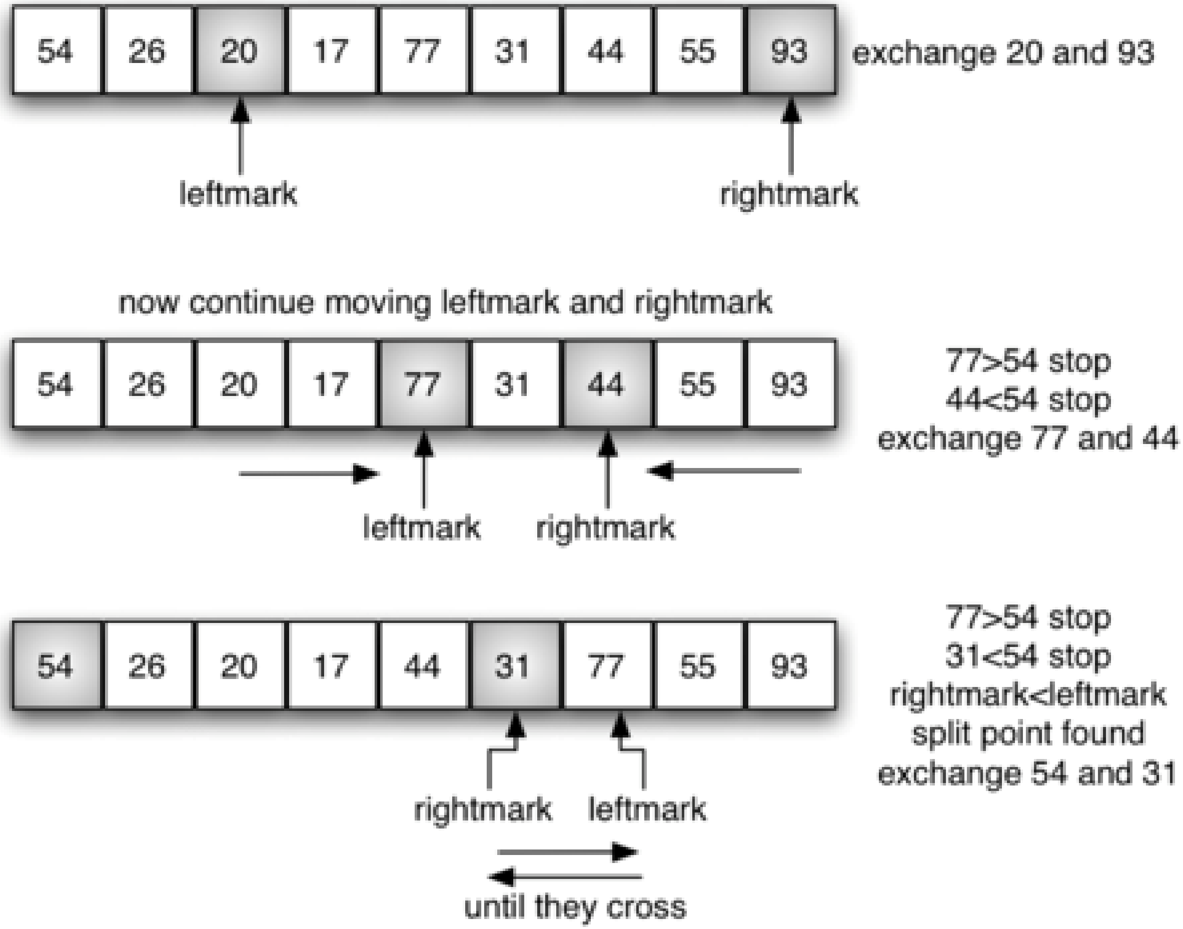
leftmark rightmark

54	26	93	17	77	31	44	55	20

now rightmark
20<54 stop

leftmark rightmark

Partition: cont'd



54 26 20 17 77 31 44 55 93

exchange 20 and 93

leftmark

rightmark

now continue moving leftmark and rightmark

54 26 20 17 77 31 44 55 93

77>54 stop
44<54 stop
exchange 77 and 44

leftmark

rightmark

54 26 20 17 44 31 77 55 93

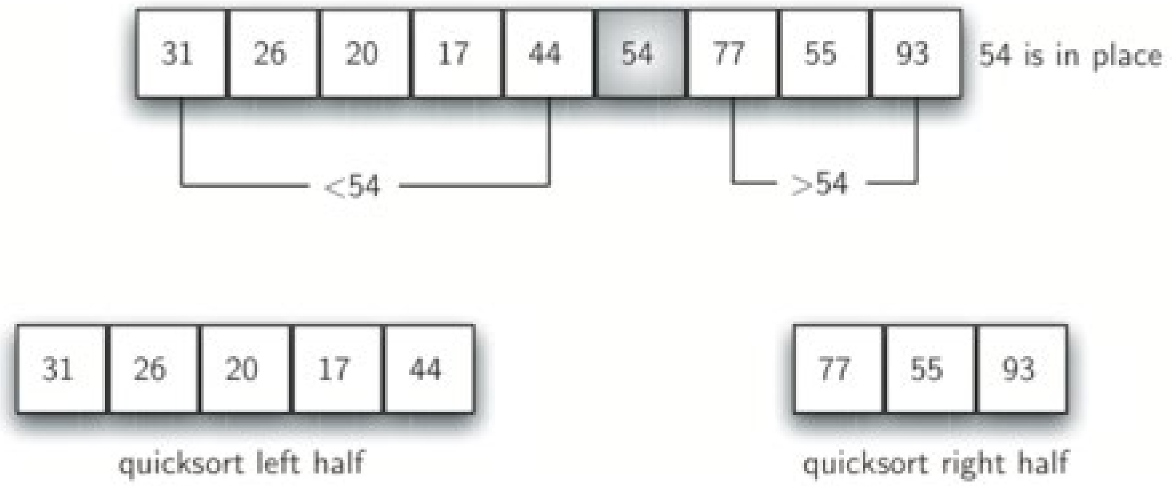
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31

rightmark

leftmark

until they cross

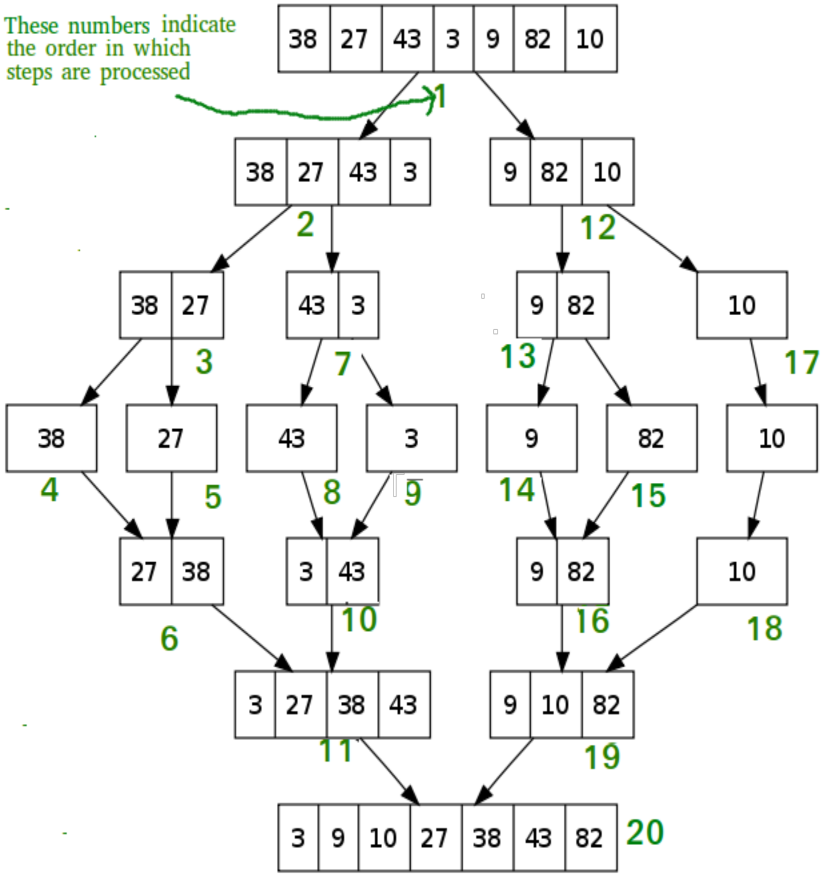
PARTITION: END



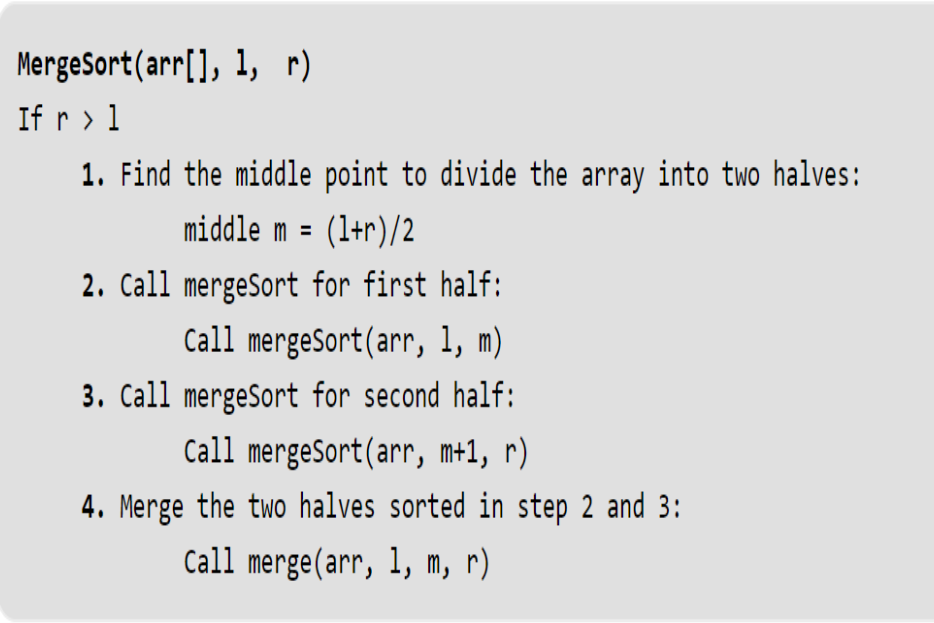
ANALYSIS OF COMPLEXITY

1. Worst case: O(N^2)
   - When the array is sorted and one choose as pivot the smallest/largest element. Then one partition is empty the other has N-1
2. Best case: O(NlogN)
   - When the pivot is the median of the array an the partitions have the same size
   - Then we have log(N) partitions on which N comparisons are applied.
3. Average case: O(NlogN)

Merge sort



Idea: Mergesort is a divide and conquer algorithm. The fundamental operations in this algorithm are dividing the array into two sub-lists with **equal length** and merging these two sorted lists.



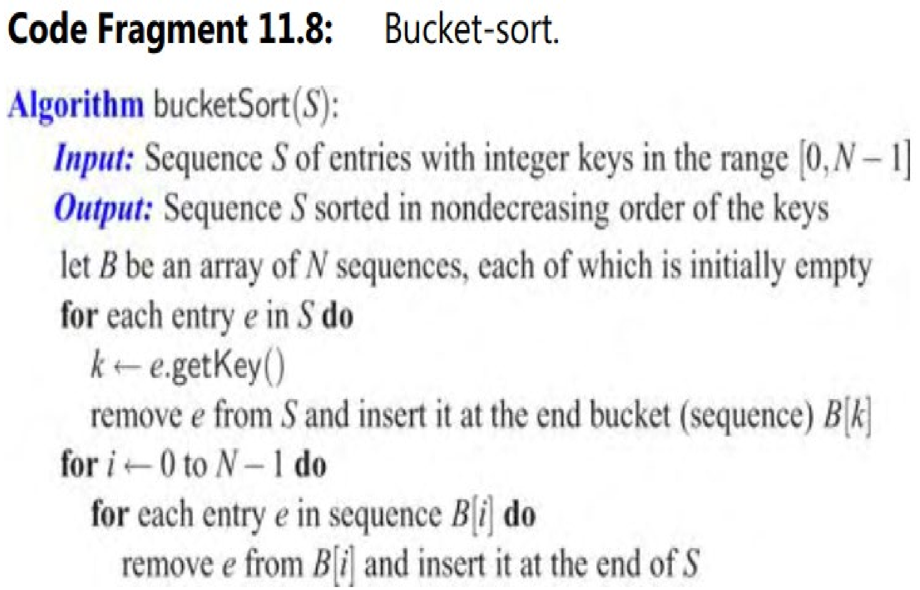
```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

Running Time of Merge-Sort

- At each level in the binary tree created for Merge-Sort O(n) time is spent (splitting and recombining sequences S_1, S_2)
- The height of the tree is O(log n) by splitting the sequences in half each time
- Therefore, the time complexity is **O(N log N)**

BUCKET-SORT

Consider a sequence S of n entries whose **keys are integers in the range [0, N−1]**, for some integer N ≥ 2, and suppose that S should be sorted according to the keys of the entries. The crucial point is that, because of the **restrictive assumption** about the format of the elements, we can avoid using comparisons



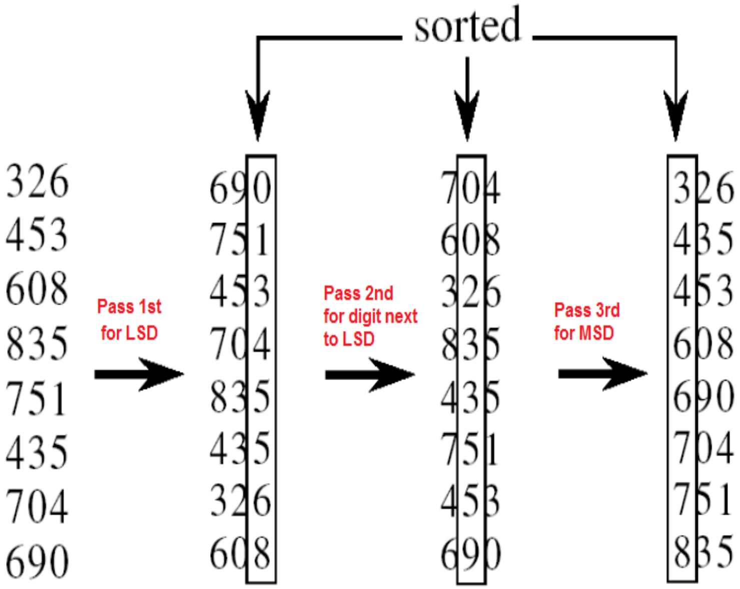
Code Fragment 11.8: Bucket-sort.

```
Algorithm bucketSort(S):
   Input: Sequence S of entries with integer keys in the range [0, N − 1]
   Output: Sequence S sorted in nondecreasing order of the keys
    let B be an array of N sequences, each of which is initially empty
    for each entry e in S do
      k ← e.getKey()
      remove e from S and insert it at the end bucket (sequence) B[k]
    for i ← 0 to N − 1 do
      for each entry e in sequence B[i] do
        remove e from B[i] and insert it at the end of S
```

Example:
For simplicity, consider the key in the range 0 to 9.
Input data: (1, v_1), (4, v_2), (1, v_3), (2, v_4), (7, v_5), (5, v_6), (2, v_7)

RADIX-SORT

We want to sort entries with **keys that are pairs (k, l), where k and l are integers in the range [0, N−1]**, for some integer N ≥ 2. In a context such as this, it is natural to define an ordering on these keys using the lexicographical (dictionary) convention, where $\boldsymbol{(k_1, l_1) < (k_2, l_2)}$ **if** $\boldsymbol{k_1 < k_2}$ **or if** $\boldsymbol{k_1 = k_2}$ **and** $\boldsymbol{l_1 < l_2}$.



PROBLEM SOLVING - PROBLEM 1

Given an array of size n, find all elements in array that appear more than n/k times. For example, if the input arrays is {3, 1, 2, 2, 1, 2, 3, 3} and k is 4, then the output should be [2, 3]. Note that size of array is 8 (or n = 8), so we need to find all elements that appear more than 2 (or 8/4) times. There are two elements that appear more than two times, 2 and 3.

Problem Solving - Problem 2

You are given a set of n real numbers and another real number x. Describe an O(*nlogn*) time algorithm that determines whether or not there exists 2 elements in S whose sum is exactly x.

Problem Solving - Problem 3

You are given an array of n+2 elements. All elements of the array are in range 1 to n. And all elements occur once except two numbers which occur twice. Find the two repeating numbers.

For example, array = {4, 2, 4, 5, 2, 3, 1} and n = 5
The above array has n + 2 = 7 elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

Problem Solving - Problem 4:

Suppose we are given an n-element sequence S such that each element in S represents a different vote for president, where each vote is given as an integer representing a particular candidate. Design an $O(n\log n)$ time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins (even if there are $O(n)$ candidates).