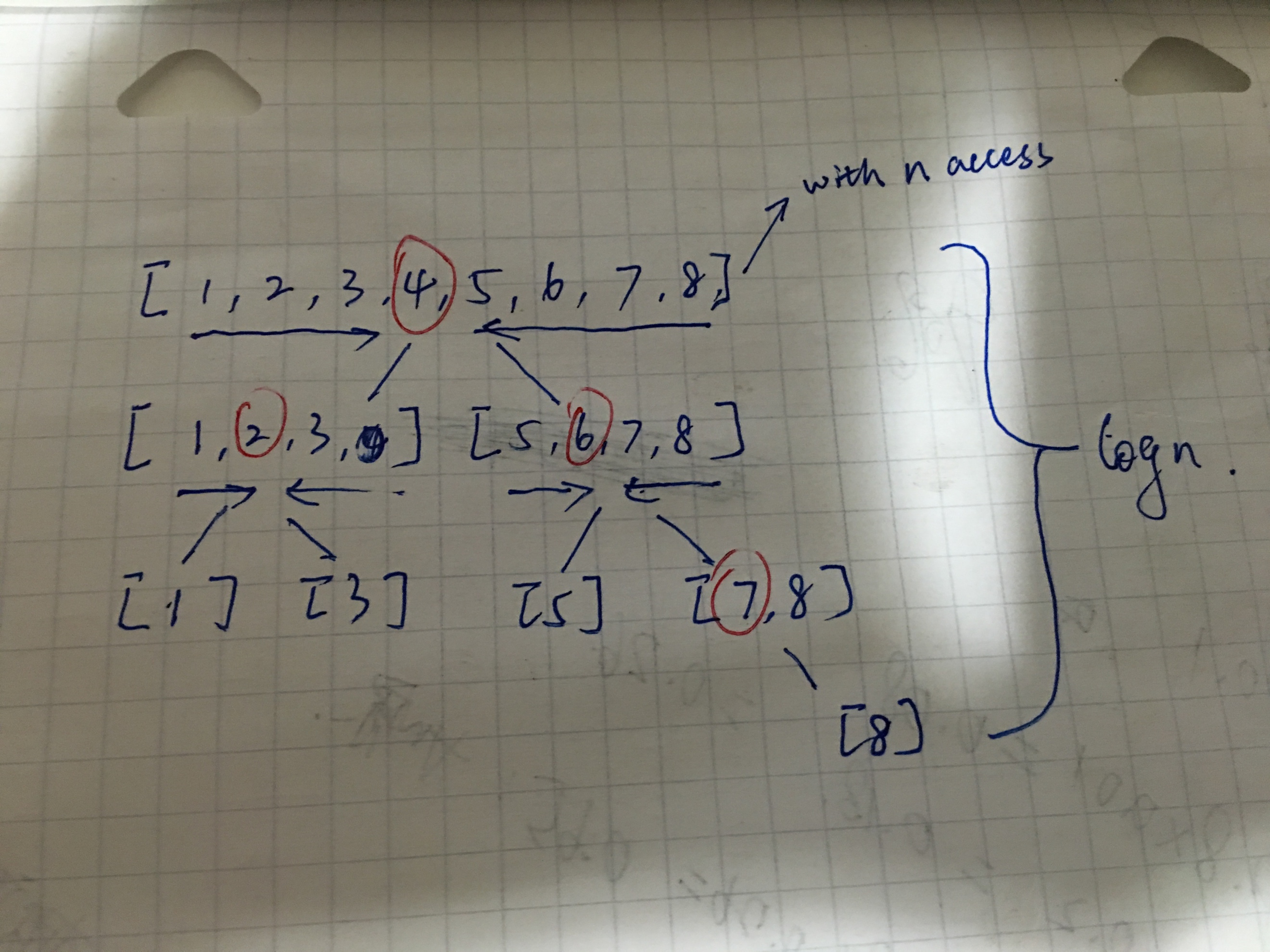
# CS-600-A Homework 5

Zijing Huang 10414952

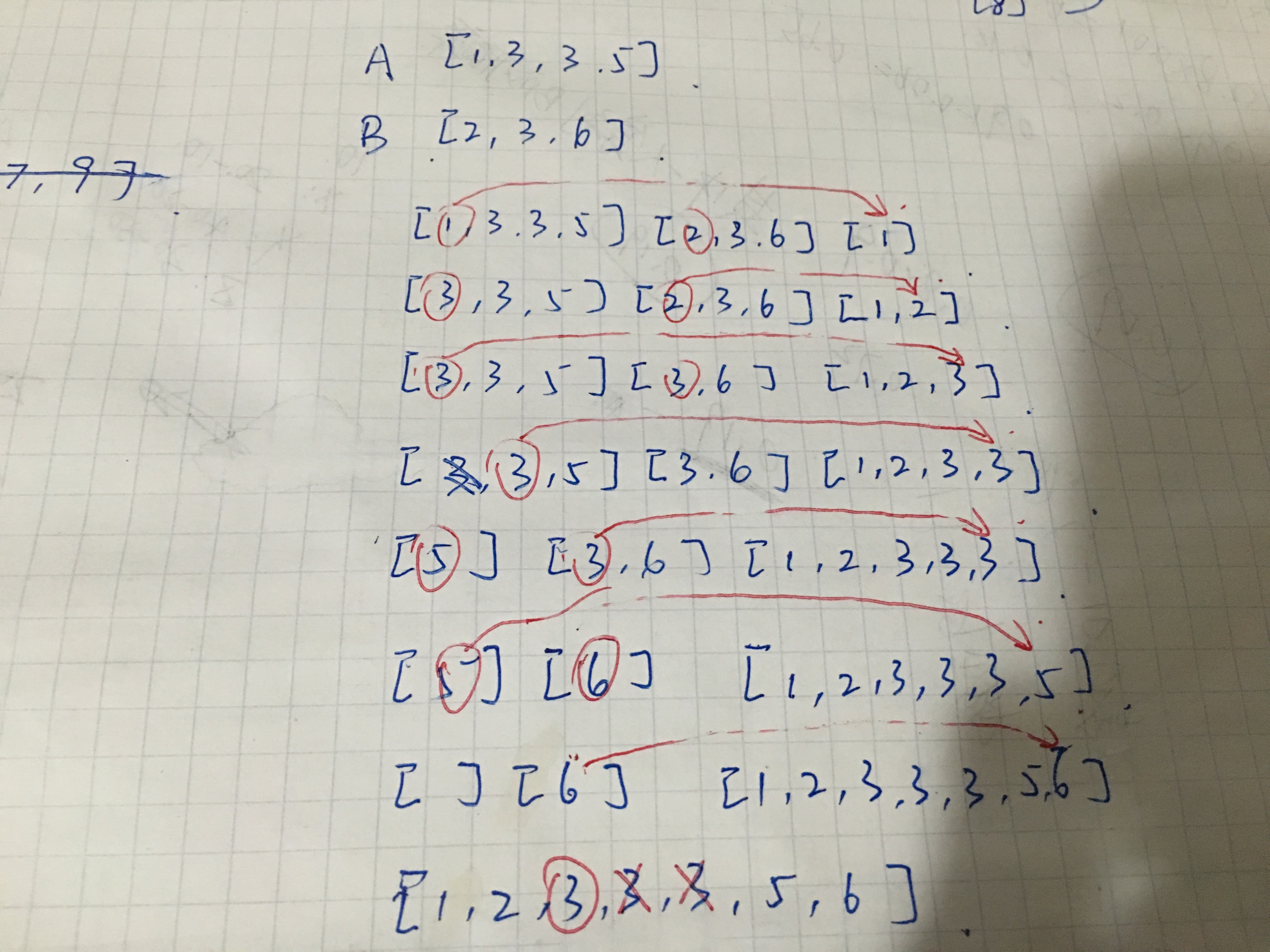
**R-8.4 Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index n/2, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted?**

The running time would be still O(nlogn). Because the pivot splits the sequence in half each time. So in each round, we still have to go through n elements. And since the height of a quick-sort tree is logn, the total running time of this version of quick-sort is O(nlogn).



**C-8.3 Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Describe an O(n)-time method for computing a sequence representing the set A ∪ B (with no duplicates).**

Firstly, since sequences A and B are sorted, we can use something like a merge sort to merge them into a new sequence. And then we traverse through the new sequence, if the next element is equal to the current element, then remove it. The time complexity would be O(n).



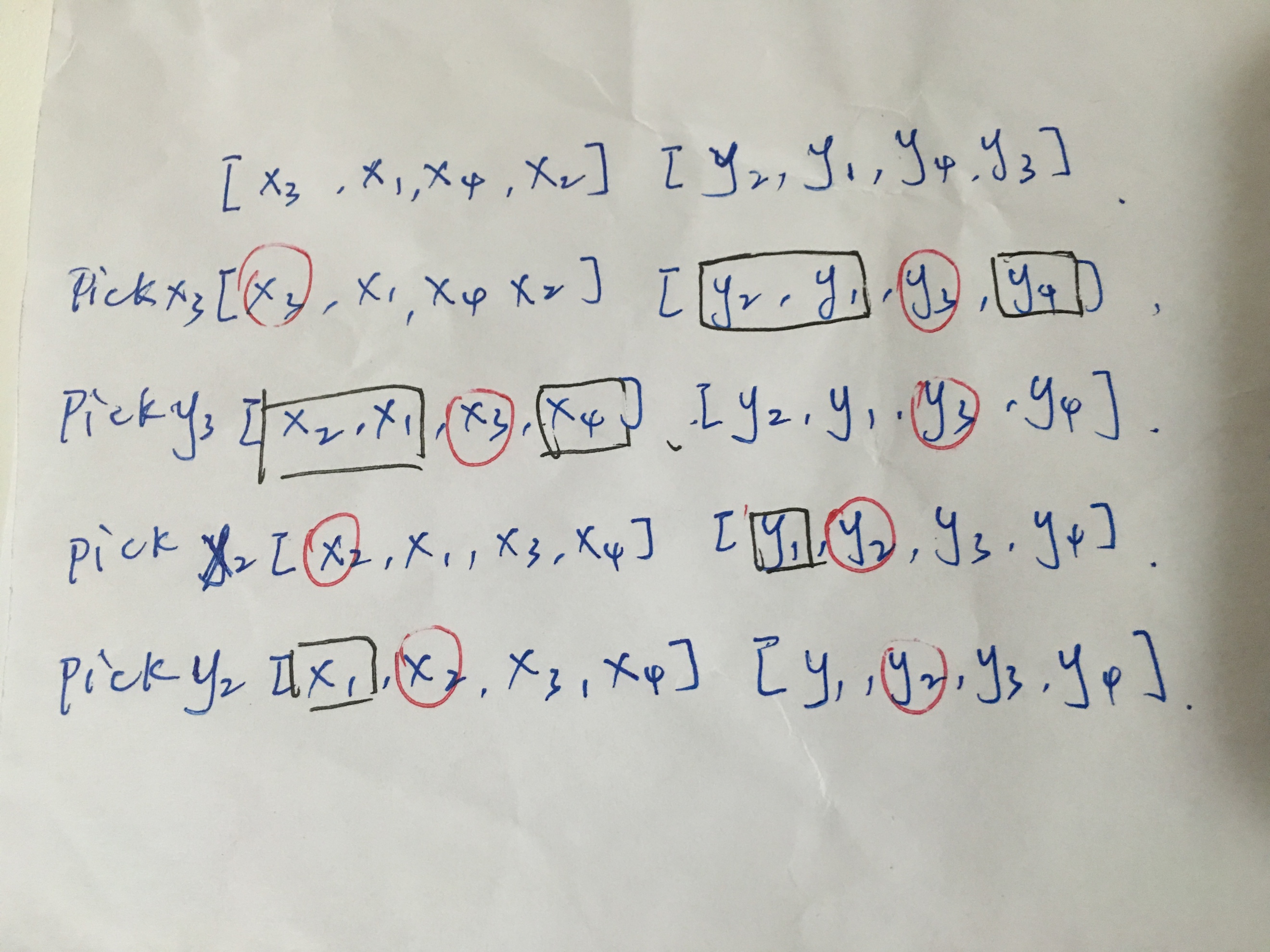
**C-8.7 Suppose we are given a sequence S of n elements, on which a total order relation is defined. Describe an efficient method for determining whether there are two equal elements in S. What is the running time of your method?**

Firstly, we can sort the sequence S. And then we scan the whole sequence, comparing the current element with the next one. If it is equal, then return true. It takes O(nlogn) time for this algorithm.

If we know the range of the elements and have a perfect hash function to put the different elements into different bucket, like a HashSet in Java. We can look up if there is an element with the same value in the HashSet before we insert a new element. If there is, return true. It would take O(n) time for this algorithm.

**A-8.4 Bob has a set, A, of n nuts and a set, B, of n bolts, such that each nut has a unique matching bolt. Unfortunately, the nuts in A all look the same, and the bolts in B all look the same as well. The only comparison that Bob can make is to take a nut-bolt pair (a, b), such that a ∈ A and b ∈ B, and test if the threads of a are larger, smaller, or a perfect match with the threads of b. Describe an efficient algorithm for Bob to match up all of his nuts and bolts. What is the running time of this algorithm?**

We can use an algorithm which is similar to quick-sort. Firstly, we randomly choose a nuts Ai that partition B into 3 parts: B0~Bi-1 < than Ai, Ai = Bi, Bi+1~Bn > Ai. After that we choose Bi and partition A into three parts as well. And then we recursively repeat it until there are less than two elements in the range. Finally, the two sets will be sorted respectively when we match up all the bolts and nuts. Because it is just like we use a quick-sort on A and B respectively, the time complexity would be O(nlogn).



**R-9.2 Describe a radix-sort method for lexicographically sorting a sequence S of triplets (k, l, m), where k, l, and m are integers in the range [0,N −1], for some N ≥ 2. How could this scheme be extended to sequences of d-tuples (k1 , k2 , . . . , kd ), where each ki is an integer in the range [0, N − 1]?**

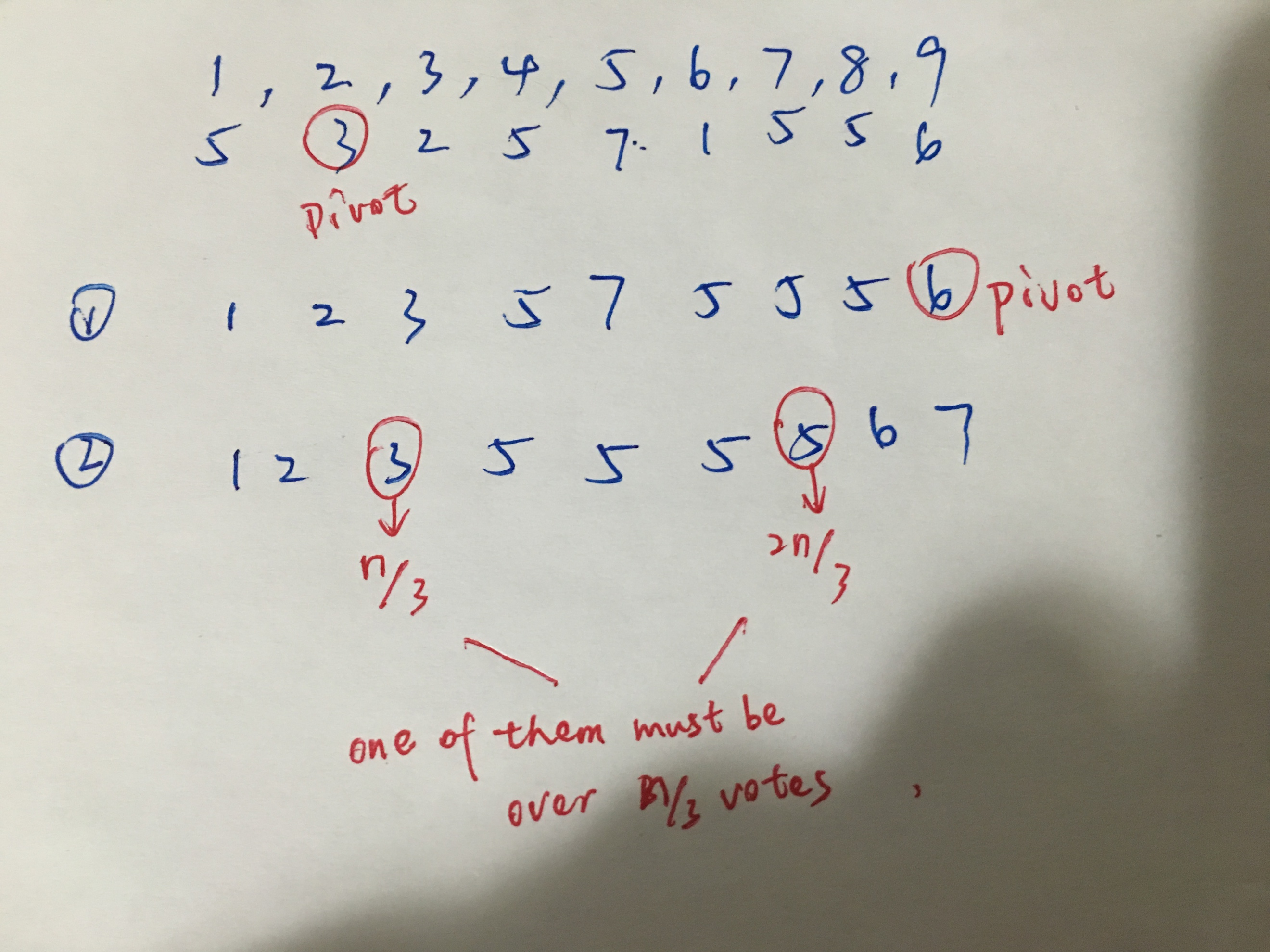
We can use a stable bucket sort to sort the sequence S, starting from m, l and then k. In the d-tuples case, we can also use a stable bucket sort, starting from kd to k1.

**C-9.5 Suppose we are given a sequence, S, of n integers in the range from 1 to n3. Give an O(n)-time method for determining whether there are two equal numbers in S.**

Since we already knew the range of n integers, we can use radix-sort in this case, which runs in O(n) time. And then we traverse the sorted sequence and see if there are two consecutive elements with the same value.

**A-9.5 Consider the election problem from the previous exercise, but now describe an algorithm running in O(n) time to determine the student numbers of every candidate that received more than n/3 votes.**

We can use a quick-select algorithm to search for the candidates of rank n/3 and 2n/3. If a candidate has more than n/3 votes, he must be in rank n/3 or rank 2n/3 or both. So we scan the sequence one more time to count exactly how many numbers of elements in the position n/3 and 2n/3 appear in the sequence. Finally, we know which is more than n/3 votes.

****