(1) List Sorting & Searching Algos

(2) Find a partner & explain how 1 algo. works to each other (2 described in total)

(3) New partner:
Write pseudocode for one of the algorithms.

(4) Next partner:
~~Either or~~ explain (in a convincing way) either
- why it terminates
- runtime

(5) Groups: (1) Algo. for finding common elements of 2 sorted arrays
(2) ~~Does~~ Pseudocode
(3) Does it terminate?
(4) What if not sorted?
(5) while loop on pg 207 - termination

30 Aug 2019

Sorting problems come in two flavors: (1.) **use sorting to make subsequent steps in an algorithm simpler**, and (2.) design a **custom sorting routine**. For the former, it's fine to use a library sort function, possibly with a custom comparator. For the latter, use a data structure like a BST, heap, or array indexed by values.

Certain problems become easier to understand, as well as solve, when the input is sorted. The most natural reason to sort is if the inputs have a **natural ordering**, and sorting can be used as a preprocessing step to **speed up searching**.

For **specialized input**, e.g., a very small range of values, or a small number of values, it's possible to sort in $O(n)$ time rather than $O(n \log n)$ time.

It's often the case that sorting can be implemented in **less space** than required by a brute-force approach.

Sometimes it is not obvious what to sort on, e.g., should a collection of intervals be sorted on starting points or endpoints? (Problem 13.5 on Page 210)

Table 13.1: Top Tips for Sorting

13.1 Compute the intersection of two sorted arrays

A natural implementation for a search engine is to retrieve documents that match the set of words in a query by maintaining an inverted index. Each page is assigned an integer identifier, its *document-ID*. An inverted index is a mapping that takes a word w and returns a sorted array of page-ids which contain w—the sort order could be, for example, the page rank in descending order. When a query contains multiple words, the search engine finds the sorted array for each word and then computes the intersection of these arrays—these are the pages containing all the words in the query. The most computationally intensive step of doing this is finding the intersection of the sorted arrays.

Write a program which takes as input two sorted arrays, and returns a new array containing elements that are present in both of the input arrays. The input arrays may have duplicate entries, but the returned array should be free of duplicates. For example, the input is $\langle 2,3,3,5,5,6,7,7,8,12\rangle$ and $\langle 5,5,6,8,8,9,10,10\rangle$, your output should be $\langle 5,6,8\rangle$.

Hint: Solve the problem if the input array lengths differ by orders of magnitude. What if they are approximately equal?

Solution: The brute-force algorithm is a "loop join", i.e., traversing through all the elements of one array and comparing them to the elements of the other array. Let m and n be the lengths of the two input arrays.

```
public static List<Integer> intersectTwoSortedArrays(List<Integer> A,
                                                      List<Integer> B) {
  List<Integer> intersectionAB = new ArrayList<>();
  for (int i = 0; i < A.size(); ++i) {
    if ((i == 0 || A.get(i) != A.get(i - 1)) && B.contains(A.get(i))) {
      intersectionAB.add(A.get(i));
    }
  }
  return intersectionAB;
}
```

The brute-force algorithm has $O(mn)$ time complexity.

Since both the arrays are sorted, we can make some optimizations. First, we can iterate through the first array and use binary search in array to test if the element is present in the second array.

```
public static List<Integer> intersectTwoSortedArrays(List<Integer> A,
                                                      List<Integer> B) {
  List<Integer> intersectionAB = new ArrayList<>();
  for (int i = 0; i < A.size(); ++i) {
    if ((i == 0 || A.get(i) != A.get(i - 1))
        && Collections.binarySearch(B, A.get(i)) >= 0) {
      intersectionAB.add(A.get(i));
    }
  }
  return intersectionAB;
}
```

The time complexity is $O(m \log n)$, where m is the length of the array being iterated over. We can further improve our run time by choosing the shorter array for the outer loop since if n is much smaller than m, then $n \log(m)$ is much smaller than $m \log(n)$.

This is the best solution if one set is much smaller than the other. However, it is not the best when the array lengths are similar because we are not exploiting the fact that both arrays are sorted. We can achieve linear runtime by simultaneously advancing through the two input arrays in increasing order. At each iteration, if the array elements differ, the smaller one can be eliminated. If they are equal, we add that value to the intersection and advance both. (We handle duplicates by comparing the current element with the previous one.) For example, if the arrays are $A = \langle 2,3,3,5,7,11 \rangle$ and $B = \langle 3,3,7,15,31 \rangle$, then we know by inspecting the first element of each that 2 cannot belong to the intersection, so we advance to the second element of A. Now we have a common element, 3, which we add to the result, and then we advance in both arrays. Now we are at 3 in both arrays, but we know 3 has already been added to the result since the previous element in A is also 3. We advance in both again without adding to the intersection. Comparing 5 to 7, we can eliminate 5 and advance to the fourth element in A, which is 7, and equal to the element that B's iterator holds, so it is added to the result. We then eliminate 11, and since no elements remain in A, we return $\langle 3,7 \rangle$.

```
public static List<Integer> intersectTwoSortedArrays(List<Integer> A,
                                                      List<Integer> B) {
  List<Integer> intersectionAB = new ArrayList<>();
  int i = 0, j = 0;
  while (i < A.size() && j < B.size()) {
    if (A.get(i) == B.get(j) && (i == 0 || A.get(i) != A.get(i - 1))) {
      intersectionAB.add(A.get(i));
      ++i;
      ++j;
    } else if (A.get(i) < B.get(j)) {
      ++i;
    } else { // A.get(i) > B.get(j).
      ++j;
    }
  }
  return intersectionAB;
}
```

Since we spend $O(1)$ time per input array element, the time complexity for the entire algorithm is $O(m+n)$.