Arrays

1. Sort an array? Bubble sort, quick sort, heap sort

2. Find total prime number, even number, odd number, armstrong number present in the array?

3. Find the highest frequency number in an array?

4. Swap every alternate index value in an array? 5 7 3 1 7 should become 7 5 1 3 7

5. Reverse the array? Don't print the array in reverse Order. Don't use 2nd array

6. Arrange the array with all even number first,then odd number in Ascending order
   Ex 5 1 6 3 2 should become 2 6 1 3 5

7. Find intersection of 2 Array?

8. Find which element is present in first array but not in 2nd array

9. Find a missing number in array of 100 numbers

10. Remove duplicates from an array of Integers, without using API methods in Java?

11. There are numbers from 1 to N in an array. Out of these, some numbers get duplicated and one is missing. The task is to find out the duplicate number.

12. **array123**: Given an array of ints, return true if the sequence of numbers 1, 2, 3 appears in the array somewhere.
    array123([1, 1, 2, 3, 1]) → true
    array123([1, 1, 2, 4, 1]) → false
    array123([1, 1, 2, 1, 2, 3]) → true

13. **arrayCount9**: Given an array of ints, return true if one of the first 4 elements in the array is a 9. The array length may be less than 4.
    arrayFront9([1, 2, 9, 3, 4]) → true
    arrayFront9([1, 2, 3, 4, 9]) → false
    arrayFront9([1, 2, 3, 4, 5]) → false

14. **array667**: Given an array of ints, return the number of times that two 6's are next to each other in the array. Also count instances where the second "6" is actually a 7.
    array667([6, 6, 2]) → 1
    array667([6, 6, 2, 6]) → 1
    array667([6, 7, 2, 6]) → 1

15. **noTriples :** Given an array of ints, we'll say that a triple is a value appearing 3 times in a row in the array. Return true if the array does not contain any triples.
    noTriples([1, 1, 2, 2, 1]) → true
    noTriples([1, 1, 2, 2, 2, 1]) → false
    noTriples([1, 1, 1, 2, 2, 2, 1]) → false

16. **matchup**: Given arrays nums1 and nums2 of the same length, for every element in nums1, consider the corresponding element in nums2 (at the same index). Return the count of the number of times that the two elements differ by 2 or less, but are not equal.
    matchUp([1, 2, 3], [2, 3, 10]) → 2
    matchUp([1, 2, 3], [2, 3, 5]) → 3
    matchUp([1, 2, 3], [2, 3, 3]) → 2

17. **modThree** : Given an array of ints, return true if the array contains either 3 even or 3 odd values all next to each other.
    modThree([2, 1, 3, 5]) → true
    modThree([2, 1, 2, 5]) → false
    modThree([2, 4, 2, 5]) → true

18. **sameEnds** : Return true if the group of N numbers at the start and end of the array are the same. For example, with {5, 6, 45, 99, 13, 5, 6}, the ends are the same for n=0 and n=2, and false for n=1 and n=3. You may assume that n is in the range 0..nums.length inclusive.
    sameEnds([5, 6, 45, 99, 13, 5, 6], 1) → false
    sameEnds([5, 6, 45, 99, 13, 5, 6], 2) → true
    sameEnds([5, 6, 45, 99, 13, 5, 6], 3) → false

19. **tripleUp** : Return true if the array contains, somewhere, three increasing adjacent numbers like .... 4, 5, 6, ... or 23, 24, 25.
    tripleUp([1, 4, 5, 6, 2]) → true
    tripleUp([1, 2, 3]) → true
    tripleUp([1, 2, 4]) → false

20. **notAlone** : We'll say that an element in an array is "alone" if there are values before and after it, and those values are different from it. Return a version of the given array where every instance of the given value which is alone is replaced by whichever value to its left or right is larger.
    notAlone([1, 2, 3], 2) → [1, 3, 3]
    notAlone([1, 2, 3, 2, 5, 2], 2) → [1, 3, 3, 5, 5, 2]
    notAlone([3, 4], 3) → [3, 4]

21. **zeroMax**: Return a version of the given array where each zero value in the array is replaced by the largest odd value to the right of the zero in the array. If there is no odd value to the right of the zero, leave the zero as a zero.

zeroMax([0, 5, 0, 3]) → [5, 5, 3, 3]
zeroMax([0, 4, 0, 3]) → [3, 4, 3, 3]
zeroMax([0, 1, 0]) → [1, 1, 0]

22. **evenOdd** : Return an array that contains the exact same numbers as the given array, but rearranged so that all the even numbers come before all the odd numbers. Other than that, the numbers can be in any order. You may modify and return the given array, or make a new array.
    evenOdd([1, 0, 1, 0, 0, 1, 1]) → [0, 0, 0, 1, 1, 1, 1]
    evenOdd([3, 3, 2]) → [2, 3, 3]
    evenOdd([2, 2, 2]) → [2, 2, 2]

23. **fizzBuzz** : Consider the series of numbers beginning at **start** and running up to but not including **end**, so for example start=1 and end=5 gives the series 1, 2, 3, 4. Return a new String[] array containing the string form of these numbers, except for multiples of 3, use "Fizz" instead of the number, for multiples of 5 use "Buzz", and for multiples of both 3 and 5 use "FizzBuzz". In Java, String.valueOf(xxx) will make the String form of an int or other type. This version is a little more complicated than the usual version since you have to allocate and index into an array instead of just printing, and we vary the start/end instead of just always doing 1..100.
    fizzBuzz(1, 6) → ["1", "2", "Fizz", "4", "Buzz"]
    fizzBuzz(1, 8) → ["1", "2", "Fizz", "4", "Buzz", "Fizz", "7"]
    fizzBuzz(1, 11) → ["1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz", "Buzz"]

24. **maxSpan** : Consider the leftmost and righmost appearances of some value in an array. We'll say that the "span" is the number of elements between the two inclusive. A single value has a span of 1. Returns the largest span found in the given array.
    maxSpan([1, 2, 1, 1, 3]) → 4
    maxSpan([1, 4, 2, 1, 4, 1, 4]) → 6
    maxSpan([1, 4, 2, 1, 4, 4, 4]) → 6

25. **fix34** : Return an array that contains exactly the same numbers as the given array, but rearranged so that every 3 is immediately followed by a 4. Do not move the 3's, but every other number may move. The array contains the same number of 3's and 4's, every 3 has a number after it that is not a 3, and a 3 appears in the array before any 4.
    fix34([1, 3, 1, 4]) → [1, 3, 4, 1]
    fix34([1, 3, 1, 4, 4, 3, 1]) → [1, 3, 4, 1, 1, 3, 4]
    fix34([3, 2, 2, 4]) → [3, 4, 2, 2]

26. **linearIn** : Given two arrays of ints sorted in increasing order, **outer** and **inner**, return true if all of the numbers in inner appear in outer. The best solution makes only a single "linear" pass of both arrays, taking advantage of the fact that both arrays are already in sorted order.
    linearIn([1, 2, 4, 6], [2, 4]) → true
    linearIn([1, 2, 4, 6], [2, 3, 4]) → false
    linearIn([1, 2, 4, 4, 6], [2, 4]) → true

27. **squareUp** : Given n>=0, create an array length n*n with the following pattern, shown here for n=3 : {0, 0, 1, 0, 2, 1, 3, 2, 1} (spaces added to show the 3 groups).
    squareUp(3) → [0, 0, 1, 0, 2, 1, 3, 2, 1]
    squareUp(2) → [0, 1, 2, 1]
    squareUp(4) → [0, 0, 0, 1, 0, 0, 2, 1, 0, 3, 2, 1, 4, 3, 2, 1]

28. **seriesUp**: Given n>=0, create an array with the pattern {1, 1, 2, 1, 2, 3, ... 1, 2, 3 .. n} (spaces added to show the grouping). Note that the length of the array will be 1 + 2 + 3 ... + n, which is known to sum to exactly n*(n + 1)/2.
    seriesUp(3) → [1, 1, 2, 1, 2, 3]
    seriesUp(4) → [1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
    seriesUp(2) → [1, 1, 2]

29. **maxMirror**: We'll say that a "mirror" section in an array is a group of contiguous elements such that somewhere in the array, the same group appears in reverse order. For example, the largest mirror section in {1, 2, 3, 8, 9, 3, 2, 1} is length 3 (the {1, 2, 3} part). Return the size of the largest mirror section found in the given array.
    maxMirror([1, 2, 3, 8, 9, 3, 2, 1]) → 3
    maxMirror([1, 2, 1, 4]) → 3
    maxMirror([7, 1, 2, 9, 7, 2, 1]) → 2

30. **countClumps**: Say that a "clump" in an array is a series of 2 or more adjacent elements of the same value. Return the number of clumps in the given array.
    countClumps([1, 2, 2, 3, 4, 4]) → 2
    countClumps([1, 1, 2, 1, 1]) → 2
    countClumps([1, 1, 1, 1, 1]) → 1

31. **scoresIncreasing** : Given an array of scores, return true if each score is equal or greater than the one before. The array will be length 2 or more.
    scoresIncreasing([1, 3, 4]) → true
    scoresIncreasing([1, 3, 2]) → false
    scoresIncreasing([1, 1, 4]) → true

32. **wordsCount**: Given an array of strings, return the count of the number of strings with the given length.
    wordsCount(["a", "bb", "b", "ccc"], 1) → 2
    wordsCount(["a", "bb", "b", "ccc"], 3) → 1
    wordsCount(["a", "bb", "b", "ccc"], 4) → 0

33. **wordsWithoutList**: Given an array of strings, return a new List (e.g. an ArrayList) where all the strings of the given length are omitted.

wordsWithoutList(["a", "bb", "b", "ccc"], 1) → ["bb", "ccc"]
wordsWithoutList(["a", "bb", "b", "ccc"], 3) → ["a", "bb", "b"]
wordsWithoutList(["a", "bb", "b", "ccc"], 4) → ["a", "bb", "b", "ccc"]

34. **copyEvens**: Given an array of positive ints, return a new array of length "count" containing the first even numbers from the original array. The original array will contain at least "count" even numbers.
    copyEvens([3, 2, 4, 5, 8], 2) → [2, 4]
    copyEvens([3, 2, 4, 5, 8], 3) → [2, 4, 8]
    copyEvens([6, 1, 2, 4, 5, 8], 3) → [6, 2, 4]

35. **matchup** : Given 2 arrays that are the same length containing strings, compare the 1st string in one array to the 1st string in the other array, the 2nd to the 2nd and so on. Count the number of times that the 2 strings are non-empty and start with the same char. The strings may be any length, including 0.
    matchUp(["aa", "bb", "cc"], ["aaa", "xx", "bb"]) → 1
    matchUp(["aa", "bb", "cc"], ["aaa", "b", "bb"]) → 2
    matchUp(["aa", "bb", "cc"], ["", "", "ccc"]) → 1

36. **scoreUp** : The "key" array is an array containing the correct answers to an exam, like {"a", "a", "b", "b"}. the "answers" array contains a student's answers, with "?" representing a question left blank. The two arrays are not empty and are the same length. Return the score for this array of answers, giving +4 for each correct answer, -1 for each incorrect answer, and +0 for each blank answer.
    scoreUp(["a", "a", "b", "b"], ["a", "c", "b", "c"]) → 6
    scoreUp(["a", "a", "b", "b"], ["a", "a", "b", "c"]) → 11
    scoreUp(["a", "a", "b", "b"], ["a", "a", "b", "b"]) → 16

37. **wordsWithout** : Given an array of strings, return a new array without the strings that are equal to the target string. One approach is to count the occurrences of the target string, make a new array of the correct length, and then copy over the correct strings.
    wordsWithout(["a", "b", "c", "a"], "a") → ["b", "c"]
    wordsWithout(["a", "b", "c", "a"], "b") → ["a", "c", "a"]
    wordsWithout(["a", "b", "c", "a"], "c") → ["a", "b", "a"]

38. **sumHeights** : We have an array of heights, representing the altitude along a walking trail. Given start/end indexes into the array, return the sum of the changes for a walk beginning at the start index and ending at the end index. For example, with the heights {5, 3, 6, 7, 2} and start=2, end=4 yields a sum of 1 + 5 = 6. The start end end index will both be valid indexes into the array with start <= end.
    sumHeights([5, 3, 6, 7, 2], 2, 4) → 6
    sumHeights([5, 3, 6, 7, 2], 0, 1) → 2
    sumHeights([5, 3, 6, 7, 2], 0, 4) → 11

39. **mergeTwo** : Start with two arrays of strings, A and B, each with its elements in alphabetical order and without duplicates. Return a new array containing the first N elements from the two arrays. The result array should be in alphabetical order and without duplicates. A and B will both have a length which is N or more. The best "linear" solution makes a single pass over A and B, taking advantage of the fact that they are in alphabetical order, copying elements directly to the new array.
    mergeTwo(["a", "c", "z"], ["b", "f", "z"], 3) → ["a", "b", "c"]
    mergeTwo(["a", "c", "z"], ["c", "f", "z"], 3) → ["a", "c", "f"]
    mergeTwo(["f", "g", "z"], ["c", "f", "g"], 3) → ["c", "f", "g"]

40. **commonTwo** : Start with two arrays of strings, a and b, each in alphabetical order, possibly with duplicates. Return the count of the number of strings which appear in both arrays. The best "linear" solution makes a single pass over both arrays, taking advantage of the fact that they are in alphabetical order.
    commonTwo(["a", "c", "x"], ["b", "c", "d", "x"]) → 2
    commonTwo(["a", "c", "x"], ["a", "b", "c", "x", "z"]) → 3
    commonTwo(["a", "b", "c"], ["a", "b", "c"]) → 3

41. **groupSum** : Given an array of ints, is it possible to choose a group of some of the ints, such that the group sums to the given target? This is a classic backtracking recursion problem. Once you understand the recursive backtracking strategy in this problem, you can use the same pattern for many problems to search a space of choices. Rather than looking at the whole array, our convention is to consider the part of the array starting at index **start** and continuing to the end of the array. The caller can specify the whole array simply by passing start as 0. No loops are needed -- the recursive calls progress down the array.
    groupSum(0, [2, 4, 8], 10) → true
    groupSum(0, [2, 4, 8], 14) → true
    groupSum(0, [2, 4, 8], 9) → false

42. **splitArray** : Given an array of ints, is it possible to divide the ints into two groups, so that the sums of the two groups are the same. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitArray(). (No loops needed.)
    splitArray([2, 2]) → true
    splitArray([2, 3]) → false
    splitArray([5, 2, 3]) → true

43. **wordLen** : Given an array of strings, return a Map<String, Integer> containing a key for every different string in the array, and the value is that string's length.
    wordLen(["a", "bb", "a", "bb"]) → {"bb": 2, "a": 1}
    wordLen(["this", "and", "that", "and"]) → {"that": 4, "and": 3, "this": 4}
    wordLen(["code", "code", "code", "bug"]) → {"code": 4, "bug": 3}

44. **pairs** : Given an array of non-empty strings, create and return a Map<String, String> as follows: for each string add its first character as a key with its last character as the value.
    pairs(["code", "bug"]) → {"b": "g", "c": "e"}
    pairs(["man", "moon", "main"]) → {"m": "n"}
    pairs(["man", "moon", "good", "night"]) → {"g": "d", "m": "n", "n": "t"}

45. **firstChar** : Given an array of non-empty strings, return a Map<String, String> with a key for every different first character seen, with the value of all the strings starting with that character appended together in the order they appear in the array.
    firstChar(["salt", "tea", "soda", "toast"]) → {"s": "saltsoda", "t": "teatoast"}
    firstChar(["aa", "bb", "cc", "aAA", "cCC", "d"]) → {"a": "aaaAA", "b": "bb", "c": "cccCC", "d": "d"}
    firstChar([]) → {}

46. **wordMultiple** : Given an array of strings, return a Map<String, Boolean> where each different string is a key and its value is true if that string appears 2 or more times in the array.
    wordMultiple(["a", "b", "a", "c", "b"]) → {"a": true, "b": true, "c": false}
    wordMultiple(["c", "b", "a"]) → {"a": false, "b": false, "c": false}
    wordMultiple(["c", "c", "c", "c"]) → {"c": true}

47. Given a matrix, find all its combinations by row. For example,
    [a, b, c]
    [d, e, f]
    [x, y, z]
    its combinations are adx, ady, adz, bdx, …. cfy, cfz

48. Find the two elements that have the smallest difference in a given array.

49. Given an array of positive and negative integers {-1,6,9,-4,-10,-9,8,8,4} (repetition allowed) sort the array in a way such that the starting from a positive number, the elements should be arranged as one positive and one negative element maintaining insertion order. First element should be starting from positive integer and the resultant array should look like {6,-1,9,-4,8,-10,8,-9,4}

50. You're given an array of integers sorted ([1,2,3,5,6,7,10]) you need to serialize and compress this array into a string (1-3, 5-7,10)