**CTCI Chapter 10: Sorting and Searching**

9.1 You are given two sorted arrays, A and B, and A has a large enough buffer at the end to hold Write a method to merge B into A in sorted order.

|  |
| --- |
| **Execution Example:**  **public** **static** **void** main(String[] args) {  **int**[] a = {2, 3, 4, 5, 6, 8, 10, 100, 0, 0, 0, 0, 0, 0};  **int**[] b = {1, 4, 7, 6, 7, 7};  merge(a, b, 8, 6);  System.*out*.println(AssortedMethods.*arrayToString*(a));  }  Yields: 1, 2, 3, 4, 4, 5, 6, 7, 6, 7, 7, 8, 10, 100 |

**High Level Design Explanation:** Review the merge method in merge sort and you should understand this quickly. This method does a similar merge but it puts the elements from largest to smallest and starts at the end and reverses its way to the beginning. This is to make use of the space at the end and also merging in place instead of adding a helper array. There is a Boolean method that determines whether we should choose to append the next largest value in A or the next largest value in B.

|  |
| --- |
| **Implementation:**    **public** **static** **void** merge(**int**[] a, **int**[] b, **int** lastA, **int** lastB) {  **int** reverseIteratorA = lastA - 1;  **int** reverseIteratorB = lastB - 1;  **int** reverseIteratorMerged = lastA + lastB - 1;    **while**(reverseIteratorB>= 0 || reverseIteratorA >= 0){  **if**(*shouldChooseA*(reverseIteratorA, reverseIteratorB, a, b)){  //copy a to merge array and decrement a  a[reverseIteratorMerged] = a[reverseIteratorA];  reverseIteratorA--;  }**else**{  //copy b to merge array and decrement b  a[reverseIteratorMerged] = b[reverseIteratorB];  reverseIteratorB--;  }  reverseIteratorMerged--;  }  }    **public** **boolean** shouldChooseA(**int** reverseIteratorA, **int** reverseIteratorB, **int**[] a, **int**[] b){  **if**(reverseIteratorA < 0){  **return** **false**; //can’t choose A  }**else** **if**(reverseIteratorB < 0){  **return** **true**; //can choose A, can’t choose B so choose A  }**else**{  **return** (a[reverseIteratorA] > b[reverseIteratorB]); //both available case  }  } |

**9.2** Write a method to sort an array of strings so that all the anagrams (example: apple and papel) are next to each other.

|  |
| --- |
| **Execution Example:**  **public** **static** **void** main(String[] args) {  String[] array = {"apple", "banana", "carrot", "ele", "duck", "papel", "tarroc", "cudk", "eel", "lee"};  System.*out*.println(AssortedMethods.*stringArrayToString*(array));  Arrays.*sort*(array, **new** AnagramComparator());  System.*out*.println(AssortedMethods.*stringArrayToString*(array));  }  **Results:** banana, carrot, tarroc, apple, papel, duck, cudk, ele, eel, lee, |

**Solution Overview:** This is just a general knowledge type problem. If you want to sort objects, you need to create a comparator class that implements comparator for the object you are sorting (in this case a string object). Then you need to override the compare function. This function returns an integer of how two objects can be compared. For example, suppose you wanted to sort students by their student numbers. If you have two students, you could return the difference between their student numbers. So for example compare (100865348, 100865349) would return -1 which means the 100865348 goes one before the 100865349. In this problem, they compare the strings based on how the strings compared to each other sorted. For example, the strings “iceman” and “cinema” would be sorted to “aceimn” and “aceimn” and when the compare function compares these, it would return zero because they are lexicographically the same and should be next to each other. This would sort the objects how you want it to.

|  |
| --- |
| **Implementation:**  **Comparator Class:**  **public** **class** AnagramComparator **implements** Comparator<String> {  **private** String sortChars(String s) {  **char**[] content = s.toCharArray();  Arrays.*sort*(content);  **return** **new** String(content);  }    **public** **int** compare(String s1, String s2) {  **return** sortChars(s1).compareTo(sortChars(s2));  }  }  And then call this in main to sort the string array:  Arrays.*sort*(array, **new** AnagramComparator()); |

9.3 Given a sorted array of n integers that has been rotated an unknown number of times, give an O (log n) algorithm that finds an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE: **Input:** find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14)

**Output:** 8 (the index of 5 in the array)

**Step 1 - Review solving for a regular sorted integer array using Binary Search:**

|  |
| --- |
| **public** **static** **int** binarySearch(**int**[] array, **int** key) {  **int** low = 0;  **int** high = array.length - 1;  **int** mid;    **while** (low <= high) {  mid = (low + high) / 2;    **if**(key == array[mid]){  **return** mid; //check if it is mid first  }**else** **if**(*keyCouldBeInThisRange*(array, key, low, mid)){  high = mid - 1;//key not at mid, so search 1st half not including mid  }**else**{  low = mid + 1; //key not at mid & not in first 1st, so search 2nd half  }  }  **return** -1;  }    **public** **static** **boolean** keyCouldBeInThisRange(**int**[] array, **int** key, **int** low, **int** high){  **return** (!key > array[high]) && (!key < array[low]);  } |

**Step 2 - Solving for sorted but rotated array:** All you have to do is modify the “could the key be in this range” method. There are two possible cases for a rotated array. The first case is if the range does not include the drop off. For example, if you are given a rotated array with the values (15 16 19 20 25 1 3 4 5 7 10 14) and your checking if the key is in the range (0 – 4) which contains the values (15 16 19 20 25) you treat it like a normal sorted array. The condition in the if-statement checks if the range contains the big drop and if it does not, then treat it like the regular sorted array. If the range does contain the big drop, i.e. (20 25 1 3 4 5 7) then the way to check if the range could contain your key is to look at the end points. For example 25 could be in the range because it is greater than the low value 20. 5 and 7 could be in the range because it is lower than the high value. When your range contains the drop you just need one of these cases to be true.

|  |
| --- |
| **public** **static** **boolean** keyCouldBeInThisRange(**int**[] array, **int** key, **int** low, **int** high){  **if**(!(array[low] > array[high])){ //if this range does not contain the big drop  **return** (!key > array[high]) && (!key < array[low]); //treat like previous box  }**else**{  **return** (key >= array[low]) || (key <= array[high]);  }  } |

**Binary Searching Strings:** The actual binary search function is pretty much the same. You have to modify the “key could be” method. I just used “does not come after the high value” and “does not come after the low value” which is logically equivalent to the integer version. The “come after and before” methods make you be able to relate to the integer function which reduces

|  |
| --- |
| **public** **static** **int** binarySearch(String[] array, String key) {  **int** low = 0;  **int** high = array.length - 1;  **int** mid;  **while** (low <= high) {  mid = (low + high) / 2;  **if**(key == array[mid]){  **return** mid; //check if it is mid first  }**else** **if**(*keyCouldBeInThisRange*(array, key, low, mid)){  high = mid - 1;//search the first half not including mid  }**else**{  low = mid + 1; //search the second half not including mid  }  }  **return** -1;  }  **public** **static** **boolean** keyCouldBeInThisRange(String[] array, String key, **int** low, **int** high){  **return**(!*comesAfter*(key,array[high])) && (!*comesBefore*(key, array[low]));  }    **public** **static** **boolean** comesAfter(String s1, String s2){  **return** s1.compareTo(s2) > 0;  }    **public** **static** **boolean** comesBefore(String s1, String s2){  **return** s1.compareTo(s2) < 0;  } |

**9.5** Given a sorted array of strings which is interspersed with empty strings, write a meth­od to find the location of a given string.

Example: find “ball” in [“at”, “”, “”, “”, “ball”, “”, “”, “car”, “”, “”, “dad”, “”, “”] will return 4

Example: find “ballcar” in [“at”, “”, “”, “”, “”, “ball”, “car”, “”, “”, “dad”, “”, “”] will return -1

Modify “key could be in this range” method: There are a few cases to consider.

1. The first case is if neither the high element and the low element in the range are an empty strings. In this case treat it like a normal binary search for strings.
2. The next case is if the high element is an empty string. For example, it is possible that “bad” could be in the range 0 and 5. The fifth element is an empy string so you would check if “bad” is in between “at” and “ball”. So when the high element is an empty string, decrement until you either get a non empty string or you hit past the low value (in which your out of range/bounds and you should return -1).
3. The next case is if the low element is an empty string. For example, it is possible that “bad” could be in the range 1 and 7. The first element is an empy string so you would check if “bad” is in between “ball” and “car”. So when the low element is an empty string, increment until you either get a non empty string or you hit past the high value(in which your out of range/bounds and you should return -1).
4. Finally, you have to think about the extreme cases. What if you keep incrementing low or decrementing high and you never find a non empty string? This would mean that the entire array range is just empty strings. In this case, both update functions would return -1 and then the “could key be in this range” method would return false.

|  |
| --- |
| **public** **static** **boolean** keyCouldBeInThisRange(String[] array, String key, **int** low, **int** high){    **if**(!array[low].equals("") && !array[high].equals("")){  //treat like keyCouldBeInThisRange() for binary searching strings  **return**(!*comesAfter*(key,array[high])) && (!*comesBefore*(key, array[low]));  }    **if**(array[low].equals("")){  low = *updateLow*(low, high, array);  }    **if**(array[high].equals("")){  high = *updateHigh*(low, high, array);  }    **if**(high == -1 || low == -1){  //This is the extreme case where the entire range in the array is “”  **return** **false**;  }  //By now we know it is not extreme case and we fixed our ends  **return**(!*comesAfter*(key,array[high])) && (!*comesBefore*(key, array[low]));  }    **public** **static** **int** updateLow(**int** low, **int** high, String[] array){  **while**(low <= high){  //While we aren’t out of range/bounds  **if**(array[low].equals("")){  low++;  }**else**{  **return** low;  }  }  **return** -1;  }    **public** **static** **int** updateHigh(**int** low, **int** high, String[] array){  **while**(high >= low){  //While we aren’t out of range/bounds  **if**(array[high].equals("")){  high--;  }**else**{  **return** high;  }  }  **return** -1;  } |