10. Sorting and Searching Questions

Question 1: Sorted Merge. Given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B, write a method to merge B into A in sorted order.

Since A has enough buffer at the end, we won't need to allocate additional space. Our logic should involve simply comparing elements of A and B and inserting them in order. The issue is that if we insert an element into the front of A, then we'll have to shift the existing elements backwards to make room for it. It's better to insert elements into the back of the array.

The code below works from the back of A and B, compares the elements at the back of the two arrays and moving the largest elements to the back of A.

```
void merge(int[] a, int[] b, int lastA, int lastB) {
  int indexA = lastA - 1; // index of last element in array a
  int indexB = lastB - 1; // index of last element in array b

  int indexMerged = lastB + lastA - 1; // end of merged array

  // merge a and b, starting from the last element in each
  while (indexB >= 0) {
    if (indexA >= 0 && a[indexA] > b[indexB]) {
      a[indexMerged] = a[indexA]; // copy element
      indexA--;
    } else {
      a[indexMerged] = b[indexB];
      indexB--;
    }
    indexMerged--;
  }
}
```

Question 3: Search in Rotated Array. Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array.

Example:

- Input: Find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}
- Output: 8 (the index of 5 in the array)

This is a modification of a binary search problem. We can see that one half of the array must be ordered normally. We can look at the normally ordered half to determine whether we should search the left or right half.

For example, if we are searching for `5` in the array, we can look at the left element (15) and compare it to the middle element (3). Since the left element > middle element, we know that the left half is not ordered properly. And since 5 is not in between 15 and 3, we must search the right half.

```
int search(int[] a, int left, int right, int x) {
  int mid = (left + right) / 2;
  if (x == a[mid]) {
    return mid;
  }
  if (right < left) {
    return -1;
  }

  if (a[left] < a[mid]) { // left is normally ordered
    if (x >= a[left] && x < a[mid]) {
      return search(a, left, mid - 1, x);
    } else {
      return search(a, mid + 1, right, x);
    }
  } else if (a[mid] < a[left]) { // right is normally ordered
    if (x > a[mid] && x <= a[right]) {
      return search(a, mid + 1, right, x);
    } else {
      return search(a, left, mid - 1, x);
    }
  } else if (a[left] == a[mid]) { // Left or right half is all repeats
    if (a[mid] != a[right]) { // if right is different, search right
      return search(a, mid + 1, right, x);
    } else { // search both halves
      int result = search(a, left, mid - 1, x); // search left
      if (result == -1) {
        return search(a, mid + 1, right, x); // search right
      } else {
        return result;
      }
    }
  }
  return -1;
}
```

The code will run in O(log n) if all elements are unique (binary search). However, with many duplicates, it will run in O(n) since it has to search both halves.

Question 4: Sorted Search, No Size.

You are given an array-like data structure Listy which lacks a size method. It does, however, have an elementAt (i) method that returns the element at index i in 0(1) time. If i is beyond the bounds of the data structure, it returns -1. (For this reason, the data structure only supports positive integers.) Given a Listy which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

Our first thought here should be binary search. The problem here is binary search requires us to know the array length. Could we compute the length? Yes. Let's try to find the length using elementAt(i) method, increasing exponentially. We'll find the length in O(log n) time.

Once we find the length, we perform a (mostly) normal binary search. There are two tweaks:

- If the midpoint is -1, we need to treat this as a "too big" value, and search left
- If in the process, the element is bigger than the value x (the one we're searching for), we'll jump over to the binary search part early.

Question 5: Sparse Search: Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

We need to modify binary search. We just have to change the part where if we compare against mid, and mid is an empty string, we move mid to the closest non-empty string.

```
int search(String[] strings, String str, int first, int last) {
  if (first > last) {
    return -1;
  }
  int mid = (first + last) / 2;

  if (strings[mid].isEmpty()) {
    int left = mid - 1;
    int right = mid + 1;
    while (true) {
      if (left < right && right > last) {
        return -1;
      } else if (right <= last && !strings[right].isEmpty()) {
        mid = right;
        break;
      } else if (left >= first && !strings[left].isEmpty()) {
        mid = left;
        break;
      }
      right++;
      left--;
    }
  }
  if (str.equals(strings[mid])) {
    return mid;
  } else if (strings[mid].compareTo(str) < 0) {
    return search(strings, str, mid + 1, last);
  } else {
    return search(strings, str, first, mid - 1);
  }
}

int search(String[] strings, String str) {
  if (strings == null || str == null || str == "") {
    return -1;
  }
  return search(strings, str, 0, strings.length - 1);
}
```

In the worst case, you will need to look at every element in the array, which is O(n).