Sorted List Matching Problem

Given two sorted lists of distinct names, output the names common to both lists.

Perhaps the standard way to attack this problem is the following:

For each name on list #1, do the following:

a) Search for the current name in list #2.

b) If the name is found, output it.

If a list is unsorted, steps a and b may take n steps, where n is the size of the second list. Can you tell me why?

If we don't use the information that either list is sorted, then we can do a brute force solution as follows:

```
void printMatches(char list1[][SIZE],
                  char list2[][SIZE],
                  int len1, int len2) {

  int i,j;

  for (i=0; i<len1; i++) {

    for (j=0; j<len2; j++) {

      if (strcmp(list1[i],list2[j]) == 0) {
        printf("%s\n", list1[i]);
        break;
      }
    }
  }
}
```

BUT, we know that both lists are already sorted. Thus we can use a binary search in step a. Assuming that both lists are of the same size (n), then the binary search takes about log n steps, which we have to repeat n times each, for a total number of steps around nlogn, which is must better than our old solution of n^2 steps.

Roughly speaking, our code would look like this:

```
void printMatches(char list1[][SIZE],
                  char list2[][SIZE],
                  int len1, int len2) {
  int i;
  for (i=0; i<len1; i++) {
    if (binSearch(list2, len2, list1[i]))
      printf("%s\n", list1[i]);
  }
}

int binSearch(char list[][SIZE], int len,
              char name[]) {

  int low = 0, high = len-1;

  while (low <= high) {
    int mid = (low+high)/2;
    int cmp = strcmp(name, list[mid]);
    if (cmp < 0)
      high = mid-1;
    else if (cmp > 0)
      low = mid+1;
    else
      return 1;
  }
  return 0;
}
```

A natural question becomes: Can we do better? The answer is yes. What is one piece of information we have that our first algorithm does NOT assume?

That list #1 is sorted. You'll notice that our previous algorithm will work regardless of the order of the names in list #1. But, we KNOW that this list is sorted also. Can we exploit this fact so that we don't have to do a full binary search for each name?

Consider how you'd probably do this task in real life...

List #1	List #2
Adams	Boston
Bell	Davis
Davis	Duncan
Harding	Francis
Jenkins	Gamble
Lincoln	Harding
Simpson	Mason
Zoeller	Simpson

You'd read that Adams and Boston are the first names on the list. Immediately you'd know that Adams wasn't a match, and neither would any name on the list #1 alphabetically before Boston. So, you'd read Bell and go on to Davis. At this point you'd deduce that Boston wasn't on the list either, so you'd read the next name on list #2 – voila!!! A match! You'd output this name and simply repeat the same idea. In particular, what we see here is that you ONLY go forward on your list of names. And for every "step" so to speak, you will read a new name off one of the two lists. Here is a more formalized version of the algorithm:

1) Start two "markers", one for each list, at the beginning of both lists.
2) Repeat the following steps until one marker has reached the end of its list.
a) Compare the two names that the markers are pointing at.
b) If they are equal, output the name and advance BOTH markers one spot.
If they are NOT equal, simply advance the marker pointing to the name that comes earlier alphabetically one spot.

Note: No code is shown here so that you can practice writing this code on your own.

<u>Algorithm Run-Time Analysis</u>

For each loop iteration, we advance at least one marker.
The maximum number of iterations then, would be the total number of names on both lists, which is n, the length of both lists.

For each iteration, we are doing a constant amount of work. (Essentially a comparison, and/or outputting a name.)

Thus, our algorithm runs in about 2n steps – an improvement over our previous algorithm.