Sorted List Matching Problem

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Sorted List Matching Problem

- Sorted List Matching Problem
  - You are given two lists of Last Names
    - Within each list, all names are distinct
    - Also, each list is already sorted
  - Problem:
    - Output the names common to both lists

Sorted List Matching Problem

- Sorted List Matching Problem
  - Perhaps a standard way to attack this problem:
  - 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 the list is unsorted, steps a and b above may take n steps, where n is the size of the second list.
    - Who can tell us why?
  - Steps a and b are run for each of the n names in List #1, resulting in an n^2 running time.

Sorted List Matching Problem

- Sorted List Matching Problem
  - If we don't take advantage of the fact that the lists are sorted, we can do a brute force algorithm 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;
                  }
            }
      }
}
```

Sorted List Matching Problem

- Sorted List Matching Problem
  - The previous solution did NOT use the fact that the lists are already sorted.
  - We can exploit this fact by using a Binary Search in step (a)
  - So what is a Binary Search…

Sorted List Matching Problem

- Binary Search
  - a binary search is an algorithm for locating the position of an item in a sorted array.
  - The idea is simple: compare the target to the middle item in the list.
  - If the target is the same as the middle item
    - you've found the target.
  - If it's before the middle item
    - repeat this procedure on the items before the middle.
  - If it's after the middle item
    - repeat on the items after the middle.
  - The method halves the number of items to check each time
    - It runs in logarithmic time: O(log n)

Sorted List Matching Problem

- Sorted List Matching Problem
  - Remember our initial algorithm:
  - 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.
  - So we use a Binary Search in step (a)
    - Assuming both lists are the same size (n)
      - Binary search takes about log n steps
      - This has to be repeated n times
        - Meaning, for each of the n names in List #1
      - So total number of steps is n*log n, or nlogn
      - Much better than our initial solution of n^2 steps

Sorted List Matching Problem

- Sorted List Matching Problem
  - 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]);
      }
}
```

  - Now let's look at the binSearch function that we are calling within this code

Sorted List Matching Problem

- Sorted List Matching Problem

```
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;
}
```

Sorted List Matching Problem

- Sorted List Matching Problem
  - A question becomes: Can we do better?
    - The answer is YES!
  - What is the one piece of information that the last algorithm did not assume?
    - Remember, we assumed that List #2 was sorted
      - This allowed us to do the Binary search on List #2
    - But we did NOT assume that List #1 is sorted.
    - Our algorithm works regardless of the order of names in List #1
    - But since List #1 is sorted, can we exploit this fact and make a better algorithm?

Sorted List Matching Problem

- Sorted List Matching Problem
  - Consider how you would actually do this task in real life (meaning with a pencil and paper)

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

Sorted List Matching Problem

- Sorted List Matching Problem
  - Consider how you would actually do this task in real life (meaning with a pencil and paper)
    - You'd see that Adams and Boston are fist on each list
    - Immediately you'd know Adams isn't a match
      - And you'd proceed down List #1 checking names, alphabetically, before Boston (from List #2)
      - So you'd skip right past Bell, knowing it can't be a match
        - Since the first name in List #2 is Boston
      - Then you come to Davis in List #1
        - And you immediately conclude that that Boston (from List #2) couldn't be a match either
      - So you move down in List #2 to Davis, and voila!, a match
        - Davis from List #1 and Davis from List #2

Sorted List Matching Problem

- Sorted List Matching Problem
  - Consider how you would actually do this task in real life (meaning with a pencil and paper)
    - So what do we recognize from this?
    - We see that we ONLY go down on the list of names
    - And for every “step”, so to speak
      - You end up reading a new name (or more) off one of the two lists
  - So now we have a more formalized version of the algorithm…

Sorted List Matching Problem

- Sorted List Matching Problem
  - Best 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 the list
       a) Compare the two names that the markers are pointing at
       b) If they are equal,
          - Output the name and advance BOTH makers one spot
       c) If they are NOT equal,
          - Simply advance the marker pointing to the name that comes earlier, alphabetically, one spot
  - Try coding this up on your own

Sorted List Matching Problem

- Sorted List Matching Problem
  - Best Algorithm: Run-Time Analysis
    - For each loop iteration, we advance at least one marker
    - As such, the maximum number of iterations 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, this algorithm runs in about 2n steps
      - An improvement over our previous algorithm

Sorted List Matching Problem

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I