CS106B Handout #17S
Spring 2013 May 3, 2013

CS106B Practice Midterm Exam #1 Solutions

Problem One: Detecting Gerrymandering (35 Points)

The simplest solution to this problem is simply to count up how many districts were won, along with the total number of votes and votes for the given political party. Given these three values, plus the number of districts, the gerrymandering ratio can be computed directly.

```
double gerrymanderingRatio(Vector<Vector<string> >& districts, string party) {
    int districtsWon = 0;
    int votesWon = 0;
    int votesTotal = 0;
    /* Compute statistics per-district. */
    foreach (Vector<string> district in districts) {
        /* Count how many votes in this district we won. */
        int votesThisDistrict = 0;
        for (string city in district) {
            if (city == party) {
               votesThisDistrict++;
            }
        }
        /* Update total number of cities and total cities we won. */
        votesTotal += district.size();
        votesWon += votesThisDistirct;

        /* Update total districts won if we won this district. */
        if (votesThisDistrict >= district.size() / 2.0) {
            districtsWon ++;
        }
    }
    /* Compute fraction of districts we won. */
    double districtRatio = double(districtsWon) / districts.size();

    /* Compute fraction of total votes we won. */
    double totalRatio = double(votesWon) / votesTotal;

    /* Gerrymandering ratio is the ratio of these quantities. */
    return districtRatio / totalRatio;
}
```

Problem Two: Cryptoquote Assistant (40 Points)

This problem is a bit harder than it looks because there are two different pieces of information to keep track of – first, which letters in the word map to which letters in the pattern string; second, which letters in the pattern string map to which letters in the word. Forgetting to get one of these directions working correctly would mean that some words would appear to match the pattern even when they did not. For example, the word "seats" does not match the pattern "aaaaa," though if you only maintain a mapping from letters in the original word to letters in the pattern you would not detect this.

There are many ways to solve this problem and we saw a wide variety of techniques as we were grading. Here are **four** different ways to solve the problem. This first solution simply keeps track of which characters get replaced with which for the two input strings:

```
Lexicon allWordsMatching(string pattern, Lexicon& words) {
    Lexicon result;
    foreach (string word in words) {
        if (matchesPattern(word, pattern)) {
            result.add(word)
        }
    }
    return result;
}
bool matchesPattern(string word, string pattern) {
    /* Wrong length never matches. */
    if (word.length() != pattern.length()) return false;

    Map<char, char> wordToPattern;
    Map<char, char> patternToWord;

    for (int i = 0; i < word.length(); i++) {
        /* If the forward direction is mapped, make sure it matches. */
        if (wordToPattern.containsKey(word[i]) &&
            wordToPattern[word[i]] != pattern[i]) {
            return false;
        }
        /* If the reverse direction is mapped, make sure it matches. */
        if (patternToWord.containsKey(pattern[i]) &&
            patternToWord[pattern[i]] != word[i]) {
            return false;
        }

        /* Otherwise, map each direction to each other. */
        wordToPattern[word[i]] = pattern[i];
        patternToWord[pattern[i]] = word[i];
    }
    return true;
}
```

Another solution we saw was to convert each word into a "canonical" version of the word by replacing each letter with a number or symbol indicating the first time it appeared in the word. You can then just find all words of the same canonical version. This is shown here:

```
Lexicon allWordsMatching(string pattern, Lexicon& words) {
    /* Convert the pattern to a canonical form. */
    string canonicalPattern = canonicalFormOf(pattern);

    /* Transform every word to its canonical form and see if it matches. */
    Lexicon result;
    foreach (string word in words) {
        if (canonicalFormOf(word) == canonicalPattern) {
            result.add(word);
        }
    }
    return result;
}

/* Converts a string to canonical form.  There are many ways to do this; one way
 * is to replace each letter with the next letter not equal to any previously-used
 * letter.
 */
string canonicalFormOf(string word) {
    /* A map from old letters to their replacement. */
    Map<char, char> replacementMap;

    /* The next available character; initially this is 'a'. */
    char nextFreeCh = 'a';

    /* Transform the word. */
    for (int i = 0; i < word.length(); i++) {
       /* If we have never seen this letter, assign it the next replacement. */
       if (!replacementMap.containsKey(word[i])) {
          replacementMap[word[i]] = nextFreeCh;
          nextFreeCh++;
       }

       /* Replace the letter with its canonical replacement. */
       word[i] = replacementMap[word[i]];
    }

    return word;
}
```

A third solution is to find, for each letter of the word, all other letters that are the same as it, then confirm that the pattern agrees at each point. You then run this process in the opposite direction to make sure that both directions work.

```
Lexicon allWordsMatching(string pattern, Lexicon& words) {
    Lexicon result;
    foreach (string word in words) {
        if (matches(word, pattern) && matches(pattern, word)) {
            result.add(word)
        }
    }

    return result;
}

bool matches(string first, string second) {
    /* Wrong length never matches. */
    if (first.length() != second.length()) return false;

    for (int i = 0; i < first.length(); i++) {
        for (int j = 0; j < first.length(); j++) {
            /* If these characters match but are mapped to different letters,
             * we have a problem.
             */
            if (first[i] == first[j] && second[i] != second[j])
                return false;
        }
    }

    return true;
}
```

A fourth solution is to construct a map from letters to their positions, then to confirm that the two maps have the same set of values.

```
Lexicon allWordsMatching(string pattern, Lexicon& words) {
    Map<char, Set<int> > patternPositions = positionMapFor(pattern);

    Lexicon result;
    foreach (string word in words) {
        if (mapsMatch(patternPositions, positionMapFor(word))) {
            result.add(word)
        }
    }

    return result;
}

bool mapsMatch(Map<char, Set<int> > first, Map<char, Set<int> > second) {
    /* If there are different numbers of characters, we're not a match. */
    if (first.size() != second.size()) return false;

    /* For each Set in the first map, see if there's a match in the second. */
    foreach (char key1 in first) {
       bool found = false;
       foreach (char key2 in second) {
          if (first[key1] == second[key2]) {
              found = true;
              break;
          }
       }
       if (!found) return false;
    }
    return true;
}

Map<char, Set<int> > positionMapFor(string word) {
    Map<char, Set<int> > result;
    for (int i = 0; i < word.length(); i++) {
        result[word[i]] += i;
    }
    return result;
}
```

Problem Three: Karel is Blocked (35 Points)

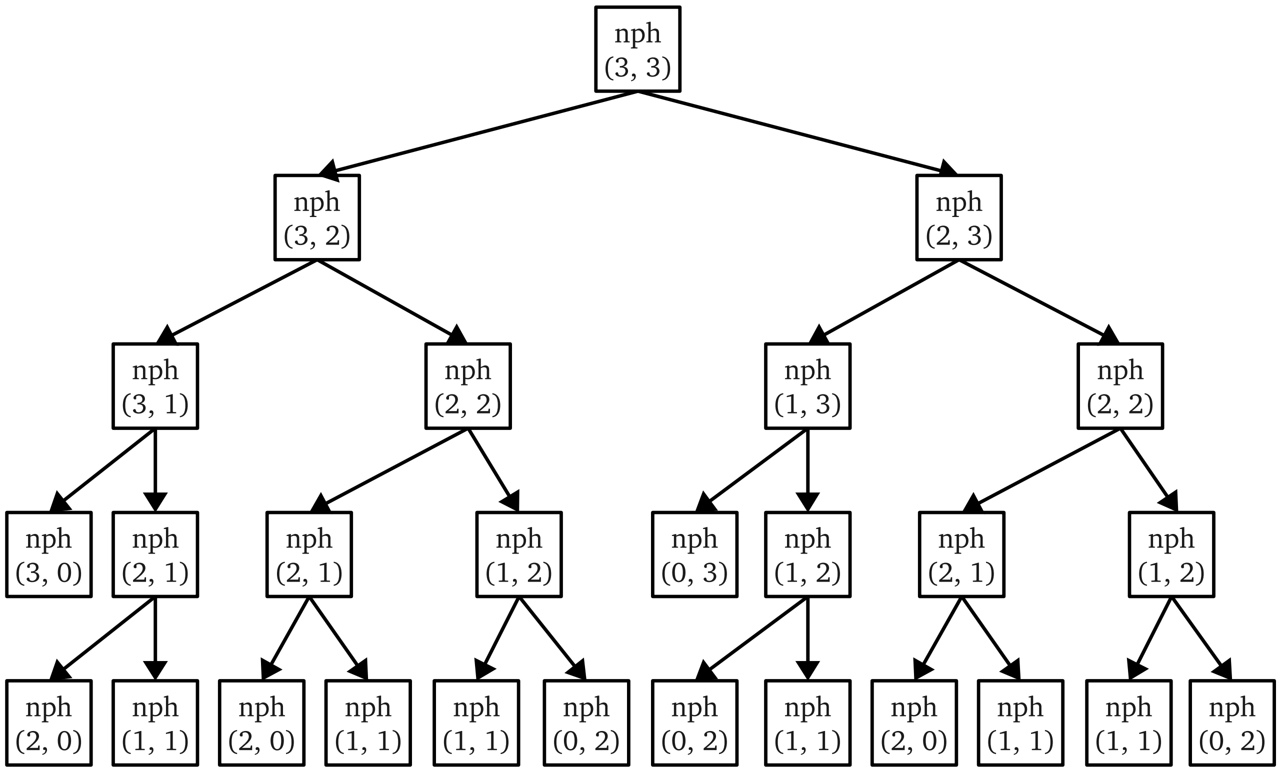
(i) When Karel Comes Marching Home (20 Points)

One way to solve this problem is to generalize the solution from the warmup problem to handle obstacles. This is shown here:

```
int numPathsHome(Grid<bool>& world, int street, int avenue) {
    /* Base Case 1: f Karel is not in the bounds of the world, there are no paths
     * back.
     */
    if (!world.inBounds(street, avenue) || street == 0 || avenue == 0)
        return 0;

    /* Base Case 2: If Karel is home, there's just one path - stay put! */
    if (street == 1 && avenue == 1)
        return 1;

    /* Base Case 3: If Karel is blocked, there are no paths back. */
    if (world[street][avenue])
        return 0;

    /* Recursive case: Count paths starting with a step to the left and a step
     * down, then combine the answers together.
     */
    return numPathsHome(world, street - 1, avenue) +
           numPathsHome(world, street, avenue - 1);
}
```

(ii) Karel Analyzes His Strategy **(15 Points)**

The answer to this question depends on your answer to part (i). Given the above solution, the recursion tree looks like this:



Many of these calls are completely the same as one another (notice that the subtrees corresponding to the calls to `numPathsHome(2, 2)` are completely identical.)

One way to address this would be to build a table storing the results of each of the recursive calls. That way, if we make a recursive call we've made before, we can avoid recomputing the answer. In particular, we could pass an extra `Grid<int>` down through the recursion. Whenever we compute a value, we can store it in the appropriate spot in the `Grid`. At the top of the function, we can see if we already have a value computed here, and (if so) can just immediately return the value.

Problem Four: Scheduling Patients (40 Points)

Here is one possible solution to the problem, which is based on the following recursive decomposition:

1. If there are no patients left to schedule, they can all be scheduled.
2. Otherwise, for each possible way of scheduling the first patient, try scheduling that patient.
3. If after making this choice the remaining patients can be scheduled, report success.
4. Otherwise, if every choice leads to failure, report that there is no schedule.

```
bool arePatientsSchedulable(Vector<int>& doctors, Vector<int>& patients) {
   /* Base case: If there are no patients, they're trivially schedulable. */
   if (patients.isEmpty()) {
      return true;
   }
   /* Recursive step: Split off the first patient and see if anyone can see
    * them.
    */
   int currentPatient = patients[0];
   Vector<int> remainingPatients = patients;
   remainingPatients.removeAt(0);

   /* For each doctor, see if that doctor can see them. */
   for (int i = 0; i < doctors.size(); i++) {
       /* If there's time, try to schedule them. */
       if (doctors[i] >= currentPatient) {
           Vector<int> updatedDoctors = doctors;
           updatedDoctors[i] -= currentPatient;

           if (arePatientsSchedulable(updatedDoctors, remainingPatients)) {
               return true;
           }
       }
   }
   /* If we're here, we cannot schedule this patient. */
   return false;
}
```

Problem Five: Modifying Merge Sort (30 Points)

In merge sort, we repeatedly split the input array into two pieces of roughly equal size, recursively sort those pieces, then merge them back together. Suppose we modify merge sort so that it still operates recursively, but chooses how it splits the array differently. Specifically, consider this algorithm:

- **Base Case**: If the array has size zero or size one, the array is already sorted.
- **Recursive Step**: Remove the last element from the array. Recursively sort the remaining array elements, then merge the sorted sequence with the sequence containing just the last element.

(i) Calculating Complexity (20 Points)

The complexity is O(n^2). At each step, when there are n elements left, the algorithm does O(n) work during the merge step in the worst case. (Depending on how you implement the merge step, this can either be O(n) work in the best case as well, or O(1) in the best case) Since there are n total elements, the amount of work done is roughly $n + (n - 1) + (n - 2) + \ldots + 2 + 1$, which is O($n^2$). This is worse than the worst-case complexity of merge sort, which is O($n \log n$).

(ii) Seem Familiar? (10 Points)

This algorithm is most similar to insertion sort. It works by growing up a sorted sequence of the elements one at a time until all of the elements are sorted.

You can also compare this algorithm to quicksort in that it has poor worst-case behavior. Depending on how you do the merge step, this could be true. However, if you implement the merge step this way, then the algorithm is almost identical to insertion sort, which also has this poor worst-case behavior.