

## CS106X Practice Exam Solution

---

### Solution 1: Acronyms

```
static void buildMap(ifstream& infile, Map<string, Vector<string> >& map) {
    while (true) {
        string line;
        getline(infile, line);
        if (infile.fail()) break;
        Vector<string> tokens = explode(line, ' ');
        string acronym = "";
        foreach (string token in tokens) {
            if (!token.empty()) {
                acronym += token[0];
            }
        }

        map[acronym] += line;
    }
}

static double percentConfusing(Map<string, Vector<string> >& map) {
    int numConfusing = 0;
    foreach (string key in map) {
        if (map[key].size() > 1) {
            numConfusing++;
        }
    }

    return double(numConfusing)/map.size();
}
```

### Solution 2: HTML Entitles

This solution relies on your being fluent in the **string** class as well as the **Map** template.

```
static string restoreString(const string& str, Map<string, char>& entityMap) {
    string translation = str;
    int start = 0;
    while (true) {
        start = translation.find('&', start);
        if (start == string::npos) return translation;
        int stop = translation.find(';', start + 1);
        string entity = translation.substr(start + 1, stop - start - 1);
        if (entityMap.containsKey(entity)) {
            translation.replace(start, stop - start + 1, 1, entityMap[entity]);
            start++;
        } else {
            start = stop + 1;
        }
    }
}
```

### Solution 3: Pascal's Travels

```

static int countPaths(Grid<int>& board, int row, int col) {
    if ((row == board.numRows() - 1) &&
        (col == board.numCols() - 1)) return 1;

    if (!board.inBounds(row, col)) return 0;
    if (board[row][col] == 0) return 0;

    int hop = board[row][col];
    board[row][col] = 0;
    int count =
        countPaths(board, row + hop, col) + countPaths(board, row - hop, col) +
        countPaths(board, row, col + hop) + countPaths(board, row, col - hop);
    board[row][col] = hop;
    return count;
}

static int countPaths(Grid<int>& board) {
    return countPaths(board, 0, 0);
}

```

### Solution 4: Longest Increasing Subsequence

```

static string longestIncreasingSubsequence(const string& prefix,
                                          const string& remaining) {
    if (remaining.empty()) return prefix;
    string rest = remaining.substr(1);
    string without = longestIncreasingSubsequence(prefix, rest);
    if (!prefix.empty() && remaining[0] <= prefix[prefix.size() - 1]) {
        return without;
    }
    string with = longestIncreasingSubsequence(prefix + remaining[0], rest);
    return with.size() > without.size() ? with : without;
}

static string longestIncreasingSubsequence(const string& str) {
    return longestIncreasingSubsequence("", str);
}

```

### Solution 5: Polydivisible Numbers

The following implementation relies on the information that we know there are a finite number of polydivisible numbers, and that a number can only be polydivisible if all of its prefixes are. The implementation is technically flawed, because the **int** can't store integers of more than 10 digits, but that's an implementation detail you didn't need to worry about (and the problem statement said so).

```

static void generatePolydivisibleNumbers(Set<int>& numbers,
                                       int value, int numDigits) {
    if (numDigits > 0 && value % numDigits > 0) return;
    if (numDigits > 0) numbers.add(value);
    int start = (numDigits == 0 ? 1 : 0);
    for (int digit = start; digit <= 9; digit++) {
        generatePolydivisibleNumbers(numbers, 10 * value + digit, numDigits + 1);
    }
}

static void generatePolydivisibleNumbers(Set<int>& numbers) {
    generatePolydivisibleNumbers(numbers, 0, 0);
}

```

## Solution 6: Sanitizing Strings

```
static string sanitize(const string& str, Vector<string>& substrings) {
    string shortest = str;
    for (int i = 0; i < substrings.size(); i++) {
        string ss = substrings[i]; // ss is short for substring
        int found = 0;
        while (true) {
            found = str.find(ss, found);
            if (found == string::npos) break;
            string reduction = str.substr(0, found) + str.substr(found + ss.size());
            string result = sanitize(reduction, substrings);
            if (result.size() < shortest.size()) {
                shortest = result;
            }
            found++;
        }
    }
    return shortest;
}
```

## Solution 7: Recursive Backtracking and Scheduling Movies

```
struct interval {
    int start;
    int end;
};

static bool intervalsOverlap(const interval& one, const interval& two) {
    return ((one.start >= two.start && one.start < two.end) ||
            (two.start >= one.start && two.start < one.end));
}

static bool canScheduleMovie(const interval& proposed,
                             Vector<interval>& scheduled) {
    for (int i = 0; i < scheduled.size(); i++) {
        const interval& scheduledInterval = scheduled[i];
        if (intervalsOverlap(proposed, scheduledInterval)) {
            return false;
        }
    }
    return true;
}

static bool canSchedule(Vector<string>& titles, int count,
                       Map<string, movie>& schedule,
                       Vector<interval>& scheduledIntervals) {

    if (titles.size() == count) return true;
    if (!schedule.containsKey(titles[count])) return false; // not required

    movie& m = schedule[titles[count]]; // reference isn't necessary
    for (int i = 0; i < m.showTimes.size(); i++) {
        interval movieInterval = { m.showTimes[i], m.showTimes[i] + m.duration };
        if (canScheduleMovie(movieInterval, scheduledIntervals)) {
            scheduledIntervals.add(movieInterval);
            if (canSchedule(titles, count + 1, schedule, scheduledIntervals))
                return true;
            scheduledIntervals.remove(scheduledIntervals.size() - 1);
        }
    }
}
```

```
    }  
  }  
  return false;  
}  
  
static bool canSchedule(Vector<string>& titles, Map<string, movie>& schedule) {  
  Vector<interval> scheduledIntervals;  
  return canSchedule(titles, 0, schedule, scheduledIntervals);  
}
```