

CS106B Practice Solution

Solution 1: Acronyms

```
void ReadIntoMap(ifstream& in, Map<Vector<string> >& map) {
    Scanner s;
    s.setSpaceOption(Scanner::IgnoreSpaces);

    while (true) {
        string line;
        getline(in, line);
        if (in.fail()) break;
        s.setInput(line);
        string acronym = "";
        while (s.hasMoreTokens())
            acronym += s.nextToken()[0];
        map[acronym].add(line); // [] creates empty vector if needed
    }
}

double PercentConfusing(Map<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

```
string RestoreString(string str, Map<char>& entityMap) {
    string translation;
    translation.reserve(str.size()); // optional, preallocates space

    // cpos stands for 'c'haracter 'pos'ition
    for (int cpos = 0; cpos < str.size(); cpos++) {
        if (str[cpos] == '&') {
            int scpos = str.find(';', cpos + 1);
            string entityCore = str.substr(cpos + 1, scpos - cpos - 1);
            if (entityMap.containsKey(entityCore)) {
                translation += entityMap[entityCore];
            } else {
                translation += '&' + entityCore + ';';
            }
            cpos = scpos;
        } else {
            translation += str[cpos];
        }
    }

    return translation;
}
```

Solution 3: Pascal's Travels

```

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

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

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

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

```

Solution 4: Longest Increasing Subsequence

```

string FindLongestIncreasingSubsequence(string increasingFixed,
                                        string remaining) {
    if (remaining.empty()) {
        return increasingFixed;
    }

    string rest = remaining.substr(1);
    string longestWithout =
        FindLongestIncreasingSubsequence(increasingFixed, rest);

    if (!increasingFixed.empty() &&
        remaining[0] <= increasingFixed[increasingFixed.size() - 1])
        return longestWithout;

    increasingFixed += remaining[0];
    string longestWith =
        FindLongestIncreasingSubsequence(increasingFixed, rest);

    return
        longestWith.size() > longestWithout.size() ?
        longestWith : longestWithout;
}

string FindLongestIncreasingSubsequence(string str) {
    return FindLongestIncreasingSubsequence("", 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 as well. The implementation is technically flawed, because the `int` can't store integers of more than 20 digits, but that's an implementation detail you didn't need to worry about (and the problem statement said so.)

```
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);
    }
}

Set<int> generatePolyDivisibleNumbers() {
    Set<int> numbers;
    generatePolyDivisibleNumbers(numbers, 0, 0);
    return numbers;
}
```

Solution 6: Sanitizing Strings

```
string Sanitize(string workingString, Vector<string>& substrings) {
    string shortest = workingString;
    for (int i = 0; i < substrings.size(); i++) {
        string substring = substrings[i];
        int found = 0;
        while (true) {
            found = workingString.find(substring, found);
            if (found == string::npos) break;
            string reduction = Sanitize(workingString.substr(0, found) +
                                     workingString.substr(found + substring.size()),
                                     substrings);
            if (reduction.size() < shortest.size()) shortest = reduction;
            found = found + 1;
        }
    }
    return shortest;
}
```

Solution 7: Recursive Backtracking and Scheduling Movies

```
struct movie {
    string name;
    int duration;           // in minutes
    Vector<int> showTimes; // each in minutes since midnight
};

struct interval {
    int start;
    int end;
};

bool canSchedule(Vector<string>& titles, Map<movie>& schedule) {
    Vector<interval> scheduledIntervals;
```

```

    return canSchedule(titles, 0, schedule, scheduledIntervals);
}

bool canSchedule(Vector<string>& titles,
                 int count,
                 Map<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.removeAt(scheduledIntervals.size() - 1);
        }
    }

    return false;
}

bool canScheduleMovie(interval proposed, Vector<interval>& scheduled) {

    for (int i = 0; i < scheduled.size(); i++) {
        interval& scheduledInterval = scheduled[i];
        if (IntervalsOverlap(proposed, scheduledInterval)) {
            return false;
        }
    }

    return true;
}

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

```