

Section #3 Solutions

Based on handouts by various current and past CS106B/X instructors and TAs.

1. longestCommonSubsequence (*backtracking*)

```
string longestCommonSubsequence(const string& s1, const string& s2) {
    if (s1.length() == 0 || s2.length() == 0) {
        return "";
    } else if (s1[0] == s2[0]) {
        return s1[0] + longestCommonSubsequence(s1.substr(1), s2.substr(1));
    } else {
        string choice1 = longestCommonSubsequence(s1, s2.substr(1));
        string choice2 = longestCommonSubsequence(s1.substr(1), s2);
        if (choice1.length() >= choice2.length()) {
            return choice1;
        } else {
            return choice2;
        }
    }
}
```

2. Flood Fill (*backtracking*)

```
int floodFillHelper(const Grid<double>& elevs, int row, int col, double prevElevation,
                    Grid<bool>& flooded) {
    if (flooded[row][col]) {
        // base case: already counted this square.
        return 0;
    }
    //solution continued on next page

    else if (elevs[row][col] > prevElevation) {
        // base case: water can't flow upward.
        return 0;
    }
    // recursive case: water flows to neighbors; total impact is this square plus impact on
    // neighboring regions
    flooded[row][col] = true;
    return 1 + floodFillHelper(elevs, row - 1, col, elevs[row][col], flooded) //north
        + floodFillHelper(elevs, row + 1, col, elevs[row][col], flooded) //south
        + floodFillHelper(elevs, row, col + 1, elevs[row][col], flooded) //east
        + floodFillHelper(elevs, row, col - 1, elevs[row][col], flooded); //west
}

int floodFill(const Grid<double>& elevs, int row, int col) {
    Grid<bool> flooded(elevs numRows(), elevs numCols(), false);
    return floodFillHelper(elevs, row, col, elevs[row][col], flooded);
}
```

3. partitionable (*backtracking*)

```
bool partitionableHelper(Vector<int>& rest, int sum1, int sum2) {
    if (rest.isEmpty()) {
        return sum1 == sum2;
    } else {
        int n = rest[0];
        rest.remove(0);
        bool answer = partitionableHelper(rest, sum1 + n, sum2)
                     || partitionableHelper(rest, sum1, sum2 + n);
        rest.insert(0, n);
        return answer;
    }
}

bool partitionable(Vector<int>& v) {
    return partitionableHelper(v, 0, 0);
}
```

4. makeChange (*backtracking*)

```
void makeChangeHelper(int amount, Vector<int>& denominations, Vector<int>& chosen) {
    if (denominations.isEmpty()) {
        if (amount == 0) {
            cout << chosen << endl;
        }
    } else {
        int denomination = denominations[0];
        denominations.remove(0);
        for (int i = 0; i <= (amount / denomination); i++) {
            chosen += i;
            makeChangeHelper(amount - (i * denomination), denominations, chosen);
            chosen.remove(chosen.size() - 1);
        }
        denominations.insert(0, denomination);
    }
}

// solution continued on next page

void makeChange(int amount, Vector<int>& denominations) {
    Vector<int> chosen;
    makeChangeHelper(amount, denominations, chosen);
}
```

5. printSquares (*backtracking*)

```
// Explore all ways to form n as a sum of squares of integers starting
// with the given min and storing the chosen results in the given set.
void printSquaresHelper(int n, int min, Set<int>& chosen) {
    if (n == 0) {
        displaySquaredSum(chosen); // base case: sum has reached n
        // See cs106x-section-3-solutions Qt project for displaySquaredSum implementation
    } else {
        // recursive case: try all combinations of every integer
        int max = (int) sqrt(n); // valid choices go up to sqrt(n)
        for (int i = min; i <= max; i++) {
            // try all combinations that include the square of this integer
            chosen.add(i);
            printSquaresHelper(n - (i * i), i + 1, chosen);
            chosen.remove(i); // backtrack
        }
    }
}

// Prints all ways to express n as a sum of squares of unique integers.
// Precondition: n >= 0
void printSquares(int n) {
    Set<int> chosen;
    printSquaresHelper(n, 1, chosen);
}
```

6. listTwiddles (*backtracking*)

```
void listTwiddlesHelper(const string& prefix, const string& str,
                       int index, const Lexicon& lex) {
    if (!lex.containsPrefix(prefix)) {
        return; // optimization; not strictly necessary but good to do
    } else if (index >= str.size()) {
        if (lex.contains(prefix)) {
            cout << prefix << endl;
        }
    } else {
        for (char ch = str[index] - 2; ch <= str[index] + 2; ch++) {
            if (isalpha(ch)) {
                listTwiddlesHelper(prefix + ch, str, index + 1, lex);
            }
        }
    }
}

void listTwiddles(const string& str, const Lexicon& lex) {
    string prefix = "";
    listTwiddlesHelper(prefix, str, /* index */ 0, lex);
}
```

7. waysToClimb (*backtracking*)

```
void waysToClimbHelper(int numStairs, Stack<int>& chosen) {
    if (numStairs <= 0) {
        cout << chosen << endl;
    } else {
        chosen.push(1);                                // choose 1
        waysToClimbHelper(numStairs - 1, chosen);      // explore
        chosen.pop();                                  // un-choose

        if (numStairs > 1) {
            chosen.push(2);                            // choose 2
            waysToClimbHelper(numStairs - 2, chosen); // explore
            chosen.pop(); // un-choose
        }
    }
}

// Precondition: numStairs >= 0
void waysToClimb(int numStairs) {
    Stack<int> chosen;
    waysToClimbHelper(numStairs, chosen);
}
```

8. XLSolutionExists (*backtracking*)

```
// see cs106x-section3-xlup-solution Qt project for a graphical implementation
bool XLHelper(const Grid<char>& grid, const string& numbers, int position,
              const coord& curr, Stack<coord>& path, Map<coord, Set<int> >& cache) {

    // base case: found a path!
    if (position == numbers.length()) return true;
    // base case: walked off the edge of the grid--no path here
    else if (!grid.inBounds(curr.row, curr.col)) return false;
    // base case: the square we're on is not the character we're looking for--no path here
    else if (grid.get(curr.row, curr.col) != numbers[position]) return false;
    // base case: already tried this (coord, position) combo--no path here
    else if (cache.containsKey(curr) && cache[curr].contains(position)) return false;

    for (Direction dir = NORTH; dir <= WEST; dir++) {
        if (XLHelper(grid, numbers, position + 1,
                     neighboringCoord(curr, dir), path, cache)) {
            path.push(curr);
            return true;
        }
    }

    cache[curr] += position; // remember failure of (coord, position) combo
    return false;
}

bool XLSolutionExists(const Grid<char>& grid, const string& numbers,
                      const coord& curr, Stack<coord>& path) {
    Map<coord, Set<int> > cache; // to speed things up
    return XLHelper(grid, numbers, /* position */ 0, curr, path, cache);
}
```