Nick Troccoli
CS 106X



# Section Handout #3: Recursive Backtracking

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

Extra practice problems: CodeStepByStep – backtracking; Textbook – chapter 9 (see 9.10, 9.11)

This week, we're ramping up from "regular" recursion to using recursive backtracking algorithms to solve more complex problems. Recall that this is our backtracking checklist from lecture. Please put it to use when writing the backtracking functions on this handout!

## Backtracking Checklist

❒ **Find what choice(s) we have at each step.** What different options do we have for the next step?

For each valid choice:

❒ **Make it and explore recursively.** Pass the information for a choice to the next recursive call(s).

❒ **Undo it after exploring.** Restore everything to the way it was before making this choice.

❒ **Find our base case(s).** What should we do when we are out of decisions?

---

### 1. longestCommonSubsequence *(backtracking)*

Write a recursive function named `longestCommonSubsequence` that returns the longest common subsequence of both strings. Recall that if a string is a subsequence of another, each of its letters occurs in the longer string in the same order, but not necessarily consecutively. For example:

```
longestCommonSubsequence("nick", "zach")          "c"
longestCommonSubsequence("nick", "106x")          ""
longestCommonSubsequence("she sells", "seashells")   "sesells"
```

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

## 2. Flood Fill *(backtracking)*

When flowing over uneven terrain, water tends to flow downward, i.e. to adjacent areas at equal or lower elevations than where it currently is. Here, we consider the case of water flowing out from a single source, like an endless spring. Write a recursive function named `floodFill` that takes a `Grid<int>` of elevations as well as `int`s storing the row and col, respectively, of the original water source and returns an `int` representing the total number of locations on the grid that the water will flow into as a result (including the original water source's location). Assume that water cannot flow diagonally through the grid. For example, if `terrain` stores the `Grid<int>` below at left, then calling `floodFill(terrain, 0, 3)` should return 7:

```
{2.5, 5.0, 3.5, 3.5},     Results in this      - - X X     Causing 7 total squares
{3.0, 1.5, 4.0, 2.0},     flooding pattern:    - - - X     to be flooded.
{0.5, 1.0, 1.5, 2.0}}     (X = flooded)        X X X X
```

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

## 3. partitionable *(backtracking)*

Write a recursive function named `partitionable` that takes a reference to a `Vector` of ints and returns true if it is possible to divide the ints into two groups such that each group has the same sum. For example, the vector `{1, 1, 2, 3, 5}` can be split into `{1, 5}` and `{1, 2, 3}`. However, the vector `{1, 4, 5, 6}` can't be split into two groups with equal sums.

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

## 4. makeChange *(backtracking)*

Write a recursive function called `makeChange` that takes in a target amount of change and a `Vector` of coin denominations (integers) and prints out every way of making that amount of change, using only the coin denominations in that vector.[1] For example, if you need to make change using only pennies, nickels, and dimes, the denominations vector would be {1, 5, 10}.

Each way of making change should be printed as the number of each coin used in the denominations vector. For example, if you were to use the above denominations vector to make change for 15 cents, the possibilities would be: `{15, 0, 0}, {10, 1, 0}, {5, 2, 0}, {5, 0, 1}, {0, 3, 0}, {0, 1, 1}`.

In these particular outputs, the first element of each group indicates the number of pennies used, the second indicates the number of nickels, and the third indicates the number of dimes.

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

[1] See https://www.youtube.com/watch?v=Y7K-zqF7KxY for reference for problem #5.

## 5. printSquares *(backtracking)*

Write a recursive function named `printSquares` that uses backtracking to find all ways to express an integer as a sum of squares of unique positive integers. For example, calling `printSquares(200)` should output:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
2^2 + 4^2 + 6^2 + 12^2
2^2 + 14^2
3^2 + 5^2 + 6^2 + 7^2 + 9^2
6^2 + 8^2 + 10^2
```

You may assume that you have access to a function called `displaySquaredSum` that prints a single line in this format when passed a `Set` of integers to be squared and summed.

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

## 6. listTwiddles *(backtracking)*

Write a recursive function called `listTwiddles` that takes a `string` and a reference to an English language `Lexicon` and prints out all the English words that are that string's twiddles. Two English words are considered *twiddles* if the letters at each position are either the same, neighboring letters, or next-to-neighboring letters. For instance, "sparks" and "snarls" are twiddles. Their second and second-to-last characters are different, but p is just two past n in the alphabet, and k comes just before l. A more dramatic example: "craggy" and "eschew". They have no letters in common, but craggy's c, r, a, g, g, and y are -2, -1, -2, -1, 2, and 2 away from the e, s, c, h, e, and w in eschew. Just to be clear, a and z are not next to each other in the alphabet; there's no wrapping around. *(Note: any word is a twiddle of itself, so it's okay to print the original string, too.)*

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

## 7. waysToClimb *(backtracking)*

In this problem, we are evaluating the following scenario: you're standing at the base of a staircase and are heading to the top. A small stride will move up one stair, and a large stride advances two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. Write a recursive function `waysToClimb` that takes a positive int representing a number of stairs and prints each unique way to climb a staircase of that height by taking strides of one or two stairs at a time. Output each way to climb the stairs on its own line, using a 1 to indicate a stride of 1 stair, and a 2 to indicate a stride of 2 stairs. For example, the call of `waysToClimb(4)` should produce the following output:

```
{1, 1, 1, 1}
{1, 1, 2}
{1, 2, 1}
{2, 1, 1}
{2, 2}
```

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)

## 8. XLSolutionExists *(backtracking) – Problem by Jerry Cain*

**See the cs106x-section3-xlup-puzzle Qt project to code this problem.**

The XL puzzle is a single-player game where one is challenged to navigate through a 7 x 7 grid of randomly placed Roman numeral digits (I, V, X, and L) and, beginning from the center square, stamp through the Roman numerals I through XL (1 through 40), each separated by a space. One can step to any neighboring square (up, down, left, or right, but not diagonally), and the same location can be used multiple times, even within the same numeral.

The puzzle itself is modeled as a `Grid<char>`, where each character in the grid is understood to be either an 'I', a 'V', an 'X', an 'L', or a space character, and the center coordinate is constrained to be a space. We provide you with the definition of a `coord`, the `Direction` type as defined in the CS106 library `"direction.h"`, and a helper function around `coord`s and `Direction`s:

| | | | | | | |
|---|---|---|---|---|---|---|
| I | I | X | V | | L | X |
| X | L | | I | X | V | I |
| | I | V | I | X | | X |
| L | I | X | | V | I | X |
| X | | X | I | X | I | |
| L | I | V | L | | X | X |
| V | I | | X | I | X | L |

```
struct coord {
    int row;
    int col;
};
coord neighboringCoord(const coord& c, Direction dir) {
    coord next = c;
    if (dir == NORTH)      next.row--;
    else if (dir == EAST)  next.col++;
    else if (dir == SOUTH) next.row++;
    else if (dir == WEST)  next.col--;
    return next;
}
```

Implement a function `XLSolutionExists` that returns a `bool` representing whether it's possible to roll over the Roman numerals I through XL, in succession, starting from the center coordinate, while respecting all of the rules. If true (it is possible), populate the path stack (passed as a reference parameter) with the sequence of coordinates you should step through. See the function header below for the full list of parameters. The `numbers` parameter represents the target string of I through XL separated by spaces.

Finally, this program will run far too slowly if it continues exploring paths on the board that have already been examined through some other branch of recursive calls. Use caching and memoization to avoid searching for a particular suffix from a particular coordinate a second time if it didn't work out the first time.

```
bool XLSolutionExists(const Grid<char>& grid, const string& numbers,
                      const coord& curr, Stack<coord>& path) {
```

**Checklist:** ❑ Identify choices ❑ Choose and explore ❑ Unchoose ❑ Base case(s)