Programming Abstractions

CS106B

Cynthia Lee
Today’s topics:

- Recursion Week Fortnight continues!
- Previous lecture:
  - Loops + recursion for *generating sequences and combinations*
- Today:
  - Continue with Friday’s topic
  - Loops + recursion for *recursive backtracking*
Generating Sequences and Combinations

Recursion + loops
Generating all possible locker lock combinations

```cpp
void generateAllCombos(int length, Vector<Vector<int>>& allCombos) {
    Vector<int> combo;
    generateAllCombos(length, allCombos, combo);
}

void generateAllCombos(int length, Vector<Vector<int>>& allCombos,
                        Vector<int>& combo) {
    // base case: this combo is full-length and ready to add
    if (combo.size() == length) {
        allCombos.add(combo);
        return;
    }

    // recursive case: add all possible next numbers to combo and continue
    for (int num = 0; num <= MAX_COMBO_NUM; num++) {
        combo.add(num);
        generateAllCombos(length, allCombos, combo);
        combo.removeBack();
    }
}
```

Stanford University
Generating all possible locker lock combinations

```cpp
void generateAllCombos(int length, Vector<Vector<int>>& allCombos) {
    Vector<int> combo;
    generateAllCombos(length, allCombos, combo);
}

void generateAllCombos(int length, Vector<Vector<int>>& allCombos, Vector<int>&& combo) {
    // base case: this combo is full-length and ready to add
    if (combo.size() == length) {
        allCombos.add(combo);
        return;
    }

    // recursive case: add all possible next numbers to combo and continue
    for (int num = 0; num <= MAX_COMBO_NUM; num++) {
        combo.add(num);
        generateAllCombos(length, allCombos, combo);
        combo.removeBack();
    }
}
```

1. Choose
2. Recurse to finish sequence
3. Un-choose

Stanford University
Backtracking
Maze solving
Backtracking

A particular behavior in recursive code where you tentatively explore many options, and recover to the nearest junction when you hit a “dead end,” so you can try a different path from that junction.

The easiest way to understand this is probably to see literal exploration and dead ends.
Maze-solving
Maze-solving

Thinking through the pseudo-code:

- Return true if there is a way to win from where we’re standing.
- Return false if there isn’t.
- *From the start position, this amounts to saying, return true if there a way to solve the whole maze, otherwise false.*
bool solveMaze(Maze & maze, Point start) {
    if (maze.isOutside(start)) return true;
    if (maze.isMarked(start)) return false;
    maze.markSquare(start);
    pause(200);
    for (Direction dir = NORTH; dir <= WEST; dir++) {
        if (!maze.wallExists(start, dir)) {
            if (solveMaze(maze, adjacentPoint(start, dir))) {
                return true;
            }
        }
    }
    maze.unmarkSquare(start);
    return false;
}
Maze-solving

In what order do we visit these spaces?

A. x1, x2, x3
B. x2, x3, x1
C. x1, x3, x2
D. We don’t visit all three
E. Other/none/more

//order of loop:
enum Direction = {NORTH, EAST, SOUTH, WEST};
The stack

What is the deepest the Stack gets (number of stack frames) during the solving of this maze?

A. Less than 5
B. 5-10
C. 11-20
D. More than 20
E. Other/none/more
Backtracking template

```cpp
bool backtrackingRecursiveFunction(args) {
    › Base case test for success: return true
    › Base case test for failure: return false
    › Loop over several options for “what to do next”:
      • Tentatively “choose” one option
      • if (“explore” with recursive call returns true) return true
      • else That tentative idea didn’t work, so “un-choose” that option,
        but don’t return false yet!--let the loop explore the other options before giving up!
    › None of the options we tried in the loop worked, so return false
}
```