Programming Abstractions

CS106B

Cynthia Lee
Today’s topics:

- Recursion Week Fortnight continues!
- Previous lecture:
  - Loops + recursion for recursive backtracking
- Today:
  - More recursive backtracking
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”:
        1. Tentatively “choose” one option
        2. if (“explore” with recursive call returns true) return true
        3. 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
}
```
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
    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();
    }
}
```

Generating all possible locker lock combinations

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

This doesn’t look for dead ends and “backtrack”—the whole point is to generate all sequences and not be picky!—but it does use loops+recursion and a choose / unchoose pattern.
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;
}
Another Backtracking Example

Periodic Table Speller
Periodic Table Speller

“gates” → “GaTeS” (Gallium, Tellurium, Sulfur)

“abstraction” → can’t do it 😞

“boat” → “BOAt” (Boron, Oxygen, Astatine)

“xe” → “Xe” (Xenon)
Thinking through approaches

Idea #1: Divide the string into pairs of characters, and test those
- Won’t work! Not all periodic table symbols are 2-letter
- There are also 1-letter and 3-letter symbols

Idea #2: read one letter at a time until you get a hit on the periodic table, then lock that in and solve for the rest
- “boat” → “BOAt” (Boron, Oxygen, Astatine)
- Try “gates” → “G “ + “ates”
  - Nope, G is not a symbol
- Try “gates” → “Ga“ + “tes”
  - Good, “Ga” is a symbol, now go on to solve “tes”
- If you can solve “tes” then the whole computation returns true, and if you can’t solve “tes” then the whole computation returns false
Thinking through approaches

Idea #2: read one letter at a time until you get a hit on the periodic table, then lock that in and solve for the rest

- Try “gates” → “G “ + “ates”
  - *Nope, G is not a symbol*
- Try “gates” → “Ga“ + “tes”
  - *Good, “Ga” is a symbol, now go on to solve “tes”*
- If you can solve “tes” then the whole computation returns true, and if you can’t solve “tes” then the whole computation returns false

Does Idea #2 work?

A. Sounds good!
B. Nope!
C. Not sure…
Greedy Algorithm design pattern

Idea #2: read one letter at a time until you get a hit on the periodic table, then lock that in and solve for the rest

- Try “gates” → “G “ + “ates”
  - *Nope, G is not a symbol*
- Try “gates” → “Ga“ + “tes”
  - *Good, “Ga” is a symbol, now go on to solve “tes”*
- If you can solve “tes” then the whole computation returns true, and if you can’t solve “tes” then the whole computation returns false

**This is a classic example of a “greedy algorithm”**
- Make choices based on what is in front of you right now
- Attitude about the future is akin to “I’ll cross that bridge when I get to it”
- Very easy + efficient when it does work, but doesn’t work for every problem
Greedy Algorithm failure

A counterexample to our Idea #2 (greedy algorithm):

- hgb

- If we look up just “H” we find it
  - The greedy algorithm makes a final choice based on what is in front of it right now, not thinking about the future
  - “H looks good, let’s lock in choice of H!”
  - But there is no way to solve “gb”
  - But returning false for the whole computation would be wrong—there is a way to solve this as HgB (Mercury, Boron)
Can the backtracking template help us?

```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”:
    1. Tentatively “choose” one option
    2. if (“explore” with recursive call returns true) return true
    3. 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
}
```
Fixing Greedy Algorithm failure with backtracking

A counterexample to our Idea #2 (greedy algorithm):

- **hgb**

  - If we look up just “H” we find it
    - The backtracking algorithm makes a *tentative* choice based on what is right in front of it
    - “H looks good, let’s **try it** and see how it goes!”
    - But there is no way to solve “gb”
  
- Since “gb” failed, rethink choice of “H”
  - (loop will go on to try 2-letter symbol “Hg”)
What are the base cases?

A. The word is empty string “”
B. The word is its original length
C. We searched the entire periodic table and can’t find word
D. Other/none/more