Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Today's topics:

- Recursion ~~Week~~ Fortnight continues!
- Today:
  - Wrap-up of Loops + recursion for *generating sequences and combinations*
  - Loops + recursion for *recursive backtracking*

Generating all possible ~~coin flip~~ die roll sequences

```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

Much nicer!!

Generating all possible ~~coin flip~~ die roll sequences

```
void generateAllSequences(int length, Vector<string>& allSequences)
{
    string sequence;
    generateAllSequences(length, allSequences, sequence);
}

void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

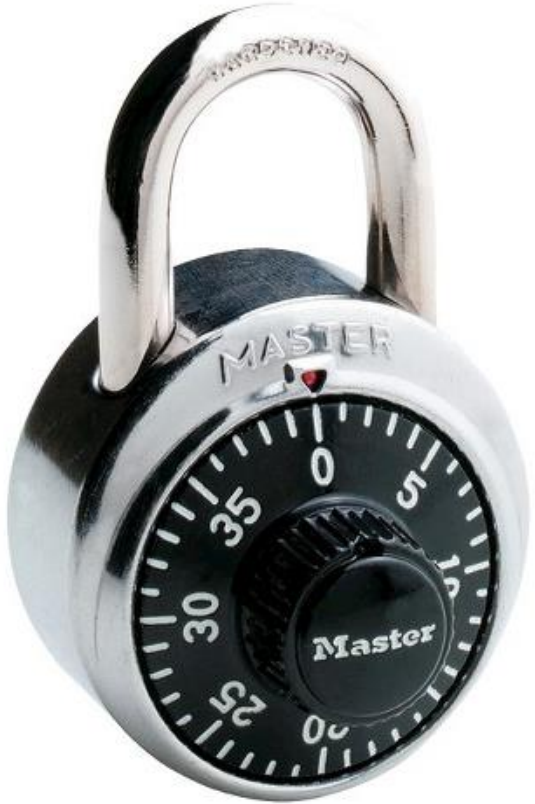
Notice that this loop **does not replace** the recursion. It just controls how many times the recursion launches.

Crack the combo lock!

TRYING TO FIND THE ONE SEQUENCE THAT WORKS

Crack the combo lock!

- You forgot the combo to your locker ☹
- It consists of 3 numbers, in the range 1-39
  - 1,1,1
  - 39,39,39
  - 2,3,4
  - 2,32,17
  - etc…
- We have no choice but to try all possible combos until we find one that unlocks the lock!
- When we find the successful combo, we save the combo in a `Vector<int>` of size 3, and return `true`. *(If we try all and it none works, the lock must be broken, return false.)*



Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!

- We'll use the die-roll code as a starting point
- Which parts we will save, and which parts need a rewrite?

```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!

Return `true/false`, so make this `bool`.

Don't need this parameter, our combo length is always 3.

Make this a pass-by-reference `Vector<int>`, so the caller gets the working combo.

die-roll code

we will save, a d a rewrite?

```
void generateAllSequences(int length, Vector<string>& allSequences, string sequence)
{
    // base case: this sequence is full-lengt
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

Don't need this collection parameter, we are only looking for one working combo.

Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!

We still want to detect when our combo is full-length (3), but it may not be the *right* full-length combo, so we need to check it.

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to add
    if (sequence.size() == length) {
        allSequences.add(sequence);
        return;
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!

We still want to detect when our combo is full-length (3), but it may not be the *right* full-length combo, so we need to check it.

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 3) {
        return tryCombo(combo);
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequence);
        sequence.erase(sequence.size() - 1);
    }
}
```

Trying all 1-39 combos sounds very similar to generating all 1-6 die roll sequences!


```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is f     ength and rea
    if (combo.size() == 3) {
        return tryCombo(combo);
    }
    // recursive cases: add 1-6 and continue
    for (int i = 1; i <= 6; i++) {
        sequence += integerToString(i);
        generateAllSequences(length, allSequences, sequ
        sequence.erase(sequence.size() - 1);
    }
}
```

We still want to loop over numbers (now 1-39).

We still want to choose a number, recursively continue generating the combo, and then “un-choose” that number before moving on to choose other numbers.

But we need to rewrite this for-loop body to take into account that a combo we try might or might not work.

Generating all possible lock sequences, to find the one successful combo


```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 3) {
        return tryCombo(combo);
    }

    // recursive cases: add 1-39 and continue
    for (int i = 1; i <= 39; i++) {
        combo += i;
        if (findCombo(combo)) {
            return true;
        }
        combo.remove(combo.size() - 1);
    }
    return false;
}
```

Choose + Recurse + Un-Choose

A COMMON RECURSIVE DESIGN PATTERN

Generating all possible coin flip sequences


```
    // Coin Fli

    // recursive cases: add H or T
    sequence += "H";
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
    sequence += "T";
    generateAllSequence  ngth, allSequences, sequence);
}
```

1. Choose an option for the next step ("H")

2. Recursion to explore more steps of the sequence

3. Un-choose that option so we can try the other option ("T") for this current step

A common design pattern in our solution: choose/unchoose

```
// Die Roll

// recursive cases: add 1-6 and continue
for (int i = 1; i <= 6; i++) {
    sequence += integerToString(i);
    generateAllSequences(length, allSequences, sequence);
    sequence.erase(sequence.size() - 1);
}
```

Choose

2. Explore

3. Un-choose

A common design pattern in our solution: choose/unchoose

```
// Combo Lock

// recursive cases: add 1-39 and continue
for (int i = 1; i <= 39; i++) {
    combo += i;
    if (findCombo(combo)) {
        return true;
    }
    sequence.remove(sequence.size() - 1);
}
```

1. Choose

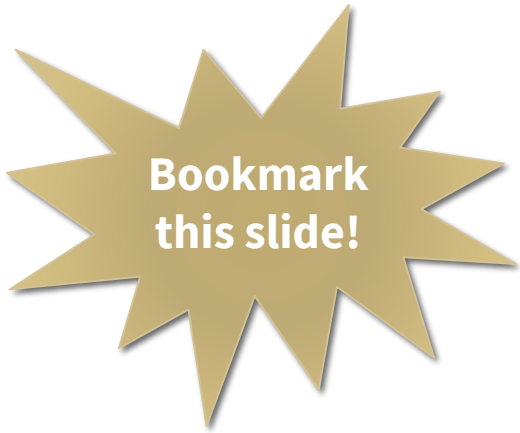
2. Explore

3. Un-choose

“Backtracking” and Choose + Recurse + Un-Choose

A SPECIAL FLAVOR OF THE COMMON RECURSIVE DESIGN PATTERN

Backtracking template



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

}

A common design pattern in our solution: Backtracking version of choose/unchoose

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 3) {
        return tryCombo(combo);
    }

    // recursive cases: add 1-39 and
    for (int i = 1; i <= 39; i++) {
        combo += i;
        if (findCombo(combo)) {
            return true;
        }
        combo.remove(combo.size() - 1);
    }
    return false;
}
```

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 giv
- None of the options we tried in the loop worked, so return false

}

Revisiting Big-O

SOME PRACTICAL TIPS

Big-O Quick Tips

- To examine program runtime, assume:
  - Single statement = 1
  - Function call = (sum of statements in function)
  - A loop of N iterations = (N * (body's runtime))

Your Turn: What is the Big-O runtime cost for this function?

```
void myFunction(int N) {
    statement1;                              // runtime = 1

    for (int i = 1; i <= N; i++) {           // runtime = N^2
        for (int j = 1; j <= N; j++) {       //    runtime = N
            statement2;                      //        runtime = 1
            statement3;                      //        runtime = 1
       }
    }

    for (int i = 1; i <= N; i++) {           // runtime = 3N
        statement4;                          //    runtime = 1
        statement5;                          //    runtime = 1
        statement6;                          //    runtime = 1
    }
}
```

Your Turn: What is the Big-O runtime cost for this function?

```
void myFunction(int N) {
    statement1;                              // runtime = 1

    for (int i = 1; i <= N; i++) {           // runtime = N^2
        for (int j = 1; j <= N; j++) {       //    runtime = N
            statement2;                      //        runtime = 1
            statement3;                      //        runtime = 1
       }
    }

    for (int i = 1; i <= N; i++) {           // runtime = 3N
        statement4;                          //    runtime = 1
        statement5;                          //    runtime = 1
        statement6;                          //    runtime = 1
    }                                        // total = 2N^2 + 3N + 1
}                                            // total = O(N^2)
```