Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Stanford University

Today’s topics:

- Recursion ~~Week~~ Fortnight continues!
- Today:
  - Review Die Roll sequence generating code from last time
    - An example of loops + recursion for *generating sequences and combinations*
  - Combination lock code
    - An example of 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++) { // style tip: should make this a const int
        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++) { // style tip:
        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 4 numbers, in the range 0-9
  - 0,0,0
  - 9,9,9
  - 2,3,4
  - 2,7,5
  - 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 will we 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);
    }
}
```

Stanford University

Rewrite Step 1: rewriting the function signature

Return true/false, so make this bool.

Don’t need this parameter, our combo length is always 4.

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

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

```
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);
    }
}
```

Rewrite Step 2: rewriting the base case

We still want to detect when our combo is full-length (4), 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);
    }
}
```

Stanford University

Rewrite Step 2: rewriting the base case

We still want to detect when our combo is full-length (4), 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() == 4) {
        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);
    }
}
```

Rewrite Step 3: rewriting the recursive case

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is f      ength and ready
    if (combo.size() == 4) {
        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 0-9).

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, and **if we find a working one, we want to exit the search early**.

Rewrite Step 3: rewriting the recursive cas

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to
    if (combo.size() == 4) { // style tip: should make this
        return tryCombo(combo);
    }

    // recursive cases: add 0-9 and continue
    for (int i = 0; i <= 9; i++) {// style tip: should make this a const int
        combo += i;
        if (findCombo(combo)) {
            return true;
        } else {
            return false;
        }
        combo.remove(combo.size() - 1);
        return false;
    }
    return false;
}
```

But we need to rewrite this for-loop body to take into account that a combo we try might or might not work, and **if we find a working one, we want to exit the search early**.

Clearly if we find a working combo, we should return true.

Do we want to return false here? (vote A)

…or somewhere else, like here or here? (vote B C)

Stanford University

Completed rewrite

```
bool findCombo(Vector<int>& combo)
{
    // base case: this sequence is full-length and ready to try on the lock!
    if (combo.size() == 4) { // style tip: should make this a const int
        return tryCombo(combo);
    }

    // recursive cases: add 0-9 and continue
    for (int i = 0; i <= 9; i++) {// style tip: should make this a const int
        combo += i;
        if (findCombo(combo)) {
            return true;
        }
        combo.remove(combo.size() - 1);
    }
    return false;
}
```

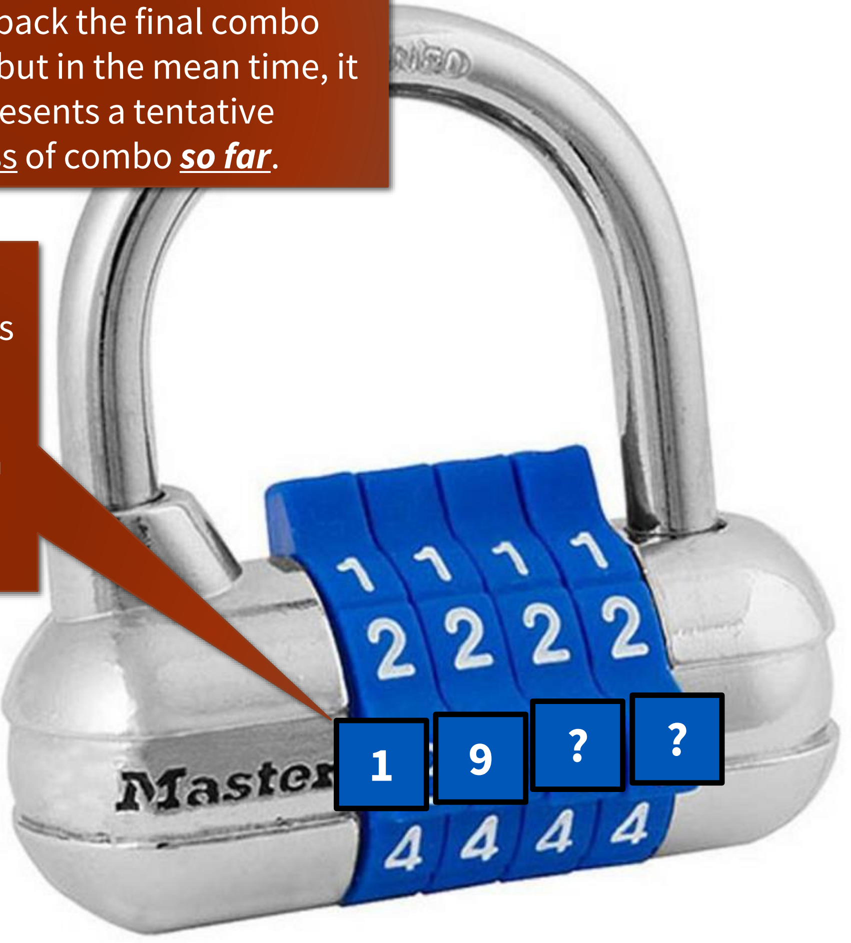
Recursive intuition

This is where we will eventually report back the final combo solution; but in the mean time, it represents a tentative <u>guess</u> of combo <u>***so far***</u>.

```
bool findCombo(Vector<int>& combo)
{
    // base case:
    if (combo.siz
        return tr
    }

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

If this function is called with the argument combo `= {1, 9}`, it is in effect saying, “Please explore all the combinations that start with 1, 9, and tell me if there is a working combo with that beginning.”



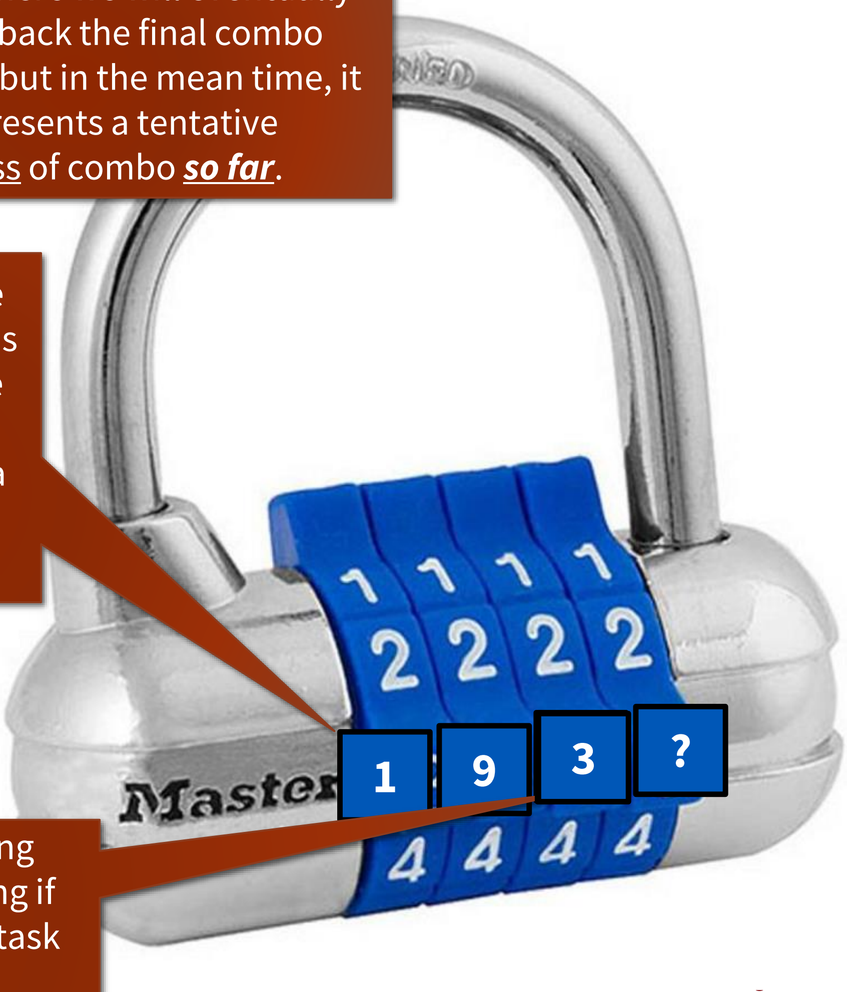
Recursive intuition

```
bool findCombo(Vector<int>& combo)
{
    // base case:
    if (combo.siz
        return tr
    }

    // recursive
    for (int i =
        combo += i;
        if (findCombo(combo)) {
            return true;
        }
        combo.remove(combo.size() - 1);
```

This is where we will eventually report back the final combo solution; but in the mean time, it represents a tentative guess of combo ***so far***.

If this function is called with the argument `combo = {1, 9}`, it is in effect saying, “Please explore all the combinations that start with 1, 9, and tell me if there is a working combo with that beginning.”



It does that exploration by saying “hm idk if there is a working combo that starts with 1, 9? Let me delegate the task of finding if there’s one that starts with 1, 9, 1. And if not, I’ll delegate the task for checking 1, 9, 2; then 1, 9, 3, etc.

Choose + Explore + 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

Stanford University

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 0-9 and continue
for (int i = 0; i <= 9; i++) {
    combo += i;
    if (findCombo(combo)) {
        return true;
    }
    sequence.remove(sequence.size() - 1);
}
```

1. Choose

2. Explore

3. Un-choose

“Backtracking” and Choose + Explore + Un-Choose

A SPECIAL FLAVOR OF THE COMMON RECURSIVE DESIGN PATTERN

Backtracking template

Bookmark this slide!

bool backtrackingRecursiveFunction*(args)* {

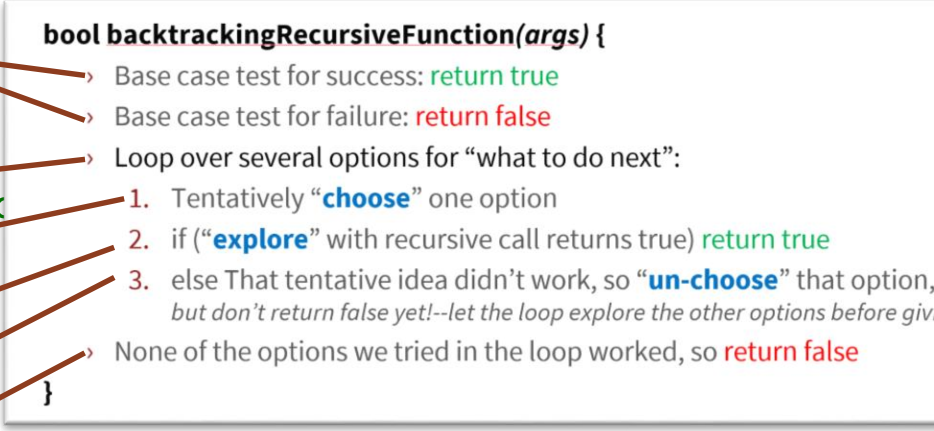
- Base case test for success: return true
- Base case test for failure: return false
- Loop over several options for "what to do next":
  1. Tentatively "**choose**" one option
  2. if ("**explore**" with recursive call returns true) return true
  3. else That tentative idea didn't work, so "**un-choose**" 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() == 4) {
        return tryCombo(combo);
    }

    // recursive cases: add 0-9 and c
    for (int i = 0; i <= 9; 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

}

Stanford University

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 = 2N^2
        for (int j = 1; j <= N; j++) {     //    runtime = 2N
            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 = 2N^2
        for (int j = 1; j <= N; j++) {        //    runtime = 2N
            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)
```