

LEC 14

CSE 123

Exhaustive Search; Recursive Backtracking

Questions during Class?
Raise hand or send here

sli.do #cse123

BEFORE WE START

Talk to your neighbors:

*What is your current favorite podcast or
YouTube channel?*

Respond on sli.do!

Instructor: Trien Vuong

TAs:	Cora	Nhan
	Jonah	Nichole
	Maitreyi	Shiven

Music: [CSE 123 26su Lecture Tunes](#)

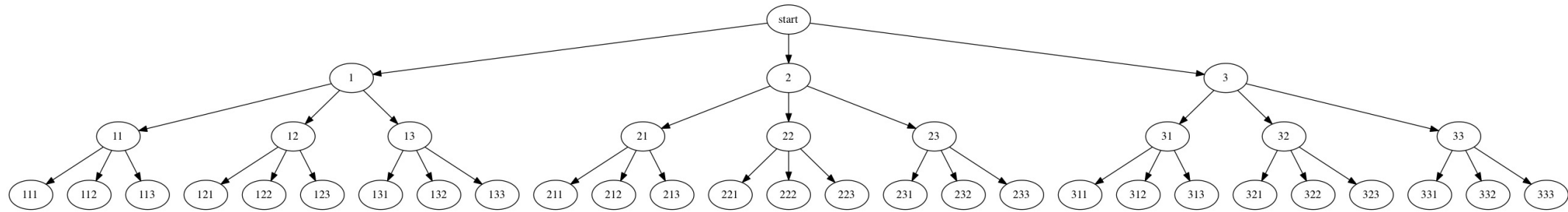
Announcements

- Creative Project 3 released, due 11:59 PM on **Friday** (8/21)
 - No resubmission for this assignment
- Resubmission Cycle 5 is due tonight at 11:59 PM
 - **P1**, C1, C2 eligible
- Final exam materials posted
- Final review session on Monday 8/17 in AND 205 from 12:00 PM to 1:00 PM
- Quiz 2 feedback will be released on Monday 8/17

Exhaustive Search Approach

- We suppose we want to explore the space of all possible solutions...
- So what do we do?
 - We "exhaustively search" through every possibility
 - We need some sort of plan or process to follow to do this programmatically
- What do we need? Recursion + some kind of accumulator
 - public / private pair

Tracing through printNums

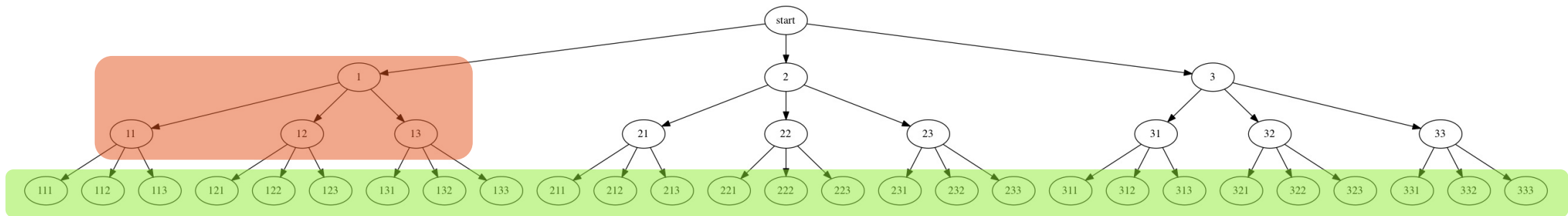


```
public static void printNums() {  
    printNums("");  
}
```

```
private static void printNums(String soFar) {  
    if (soFar.length() == 3) {  
        System.out.println(soFar);  
    } else {  
        printNums(soFar+ "1");  
        printNums(soFar+ "2");  
        printNums(soFar+ "3");  
    }  
}
```

Decision Trees

- Visual we use to help understand what our process is
 - Visualization tool, not a data structure
 - If you can draw a decision tree, you can implement exhaustive search



- Can glean important information
 - **Base case (end nodes)**
 - **Recursive case (middle nodes)**
 - "Dead end" case (more on this later...)

Exhaustive Search Pattern (search)

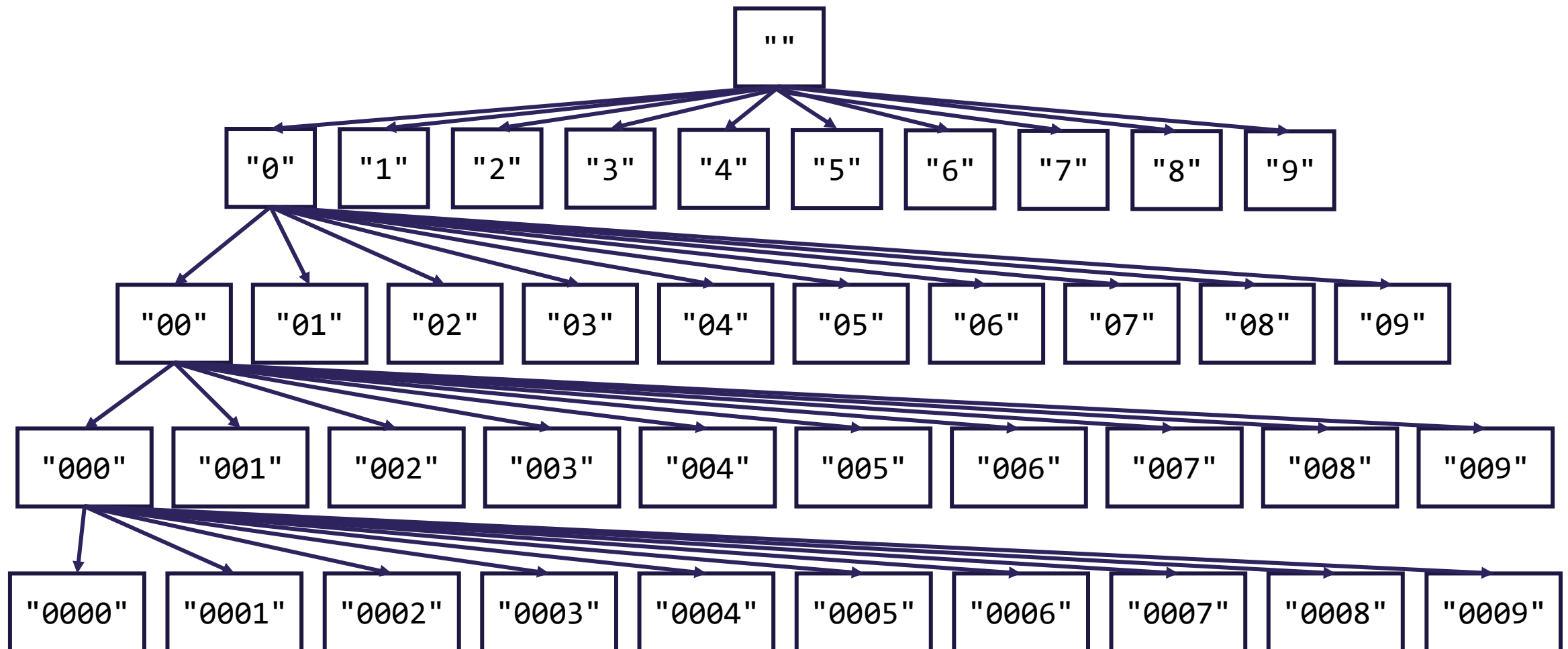
```
public static void search(input) {  
    search(input, "");  
}  
  
private static void search(input, String soFar) {  
    if (base case) {  
        // Do something with soFar (e.g. print it out)  
        System.out.println(soFar);  
    } else {  
        // Might not be a loop, but 1 recursive call for each option  
        for (each option) {  
            search(input, soFar + option);  
        }  
    }  
}
```

Exhaustive Search Pattern (printNums)

```
public static void printNums() {  
    printNums("");  
}  
  
private static void printNums(String soFar) {  
    if (soFar.length() == 3) {  
        // Do something with soFar (e.g. print it out)  
        System.out.println(soFar);  
    } else {  
        // Might not be a loop, but 1 recursive call for each option  
        for (int i = 1; i <= 3; i++) {  
            printNums(soFar + i);  
        }  
    }  
}
```

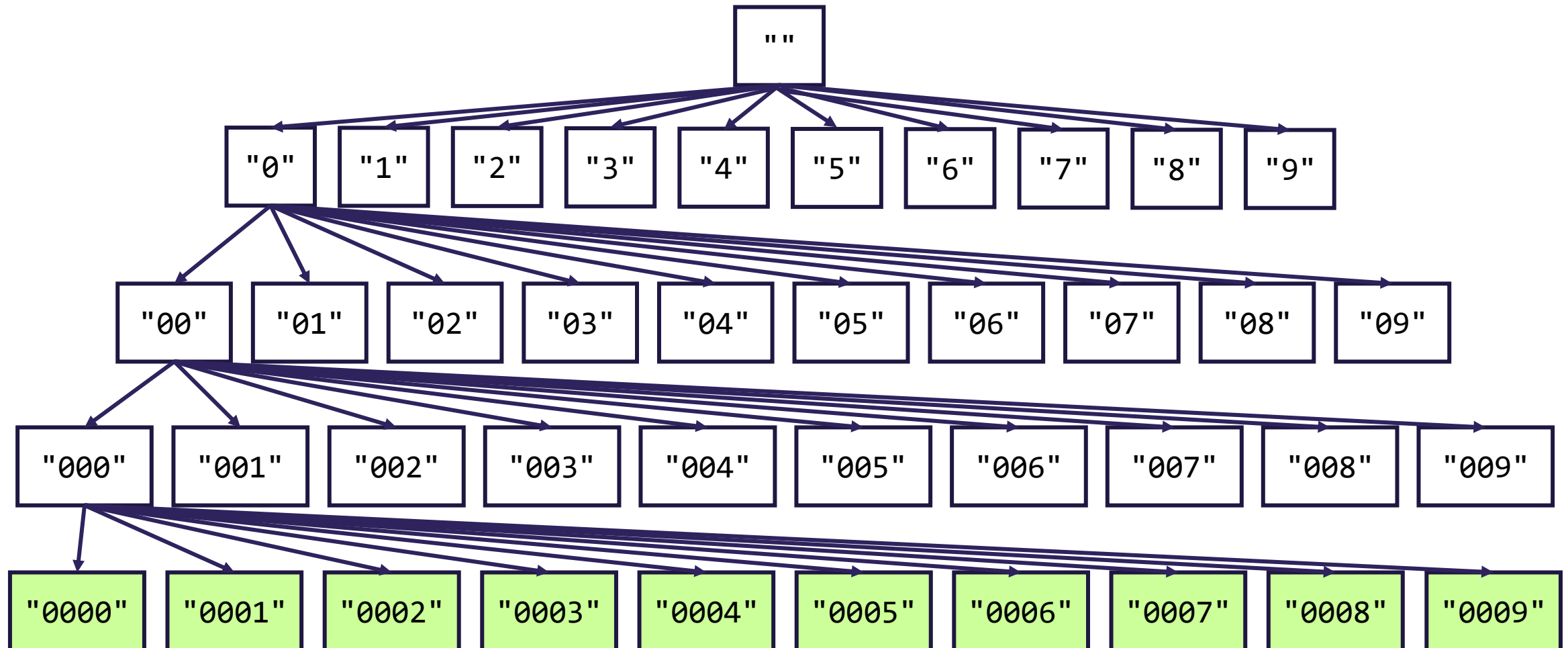
Password Cracker: Decision Tree

- Let's say we want to crack the password of a 4 digit combination lock



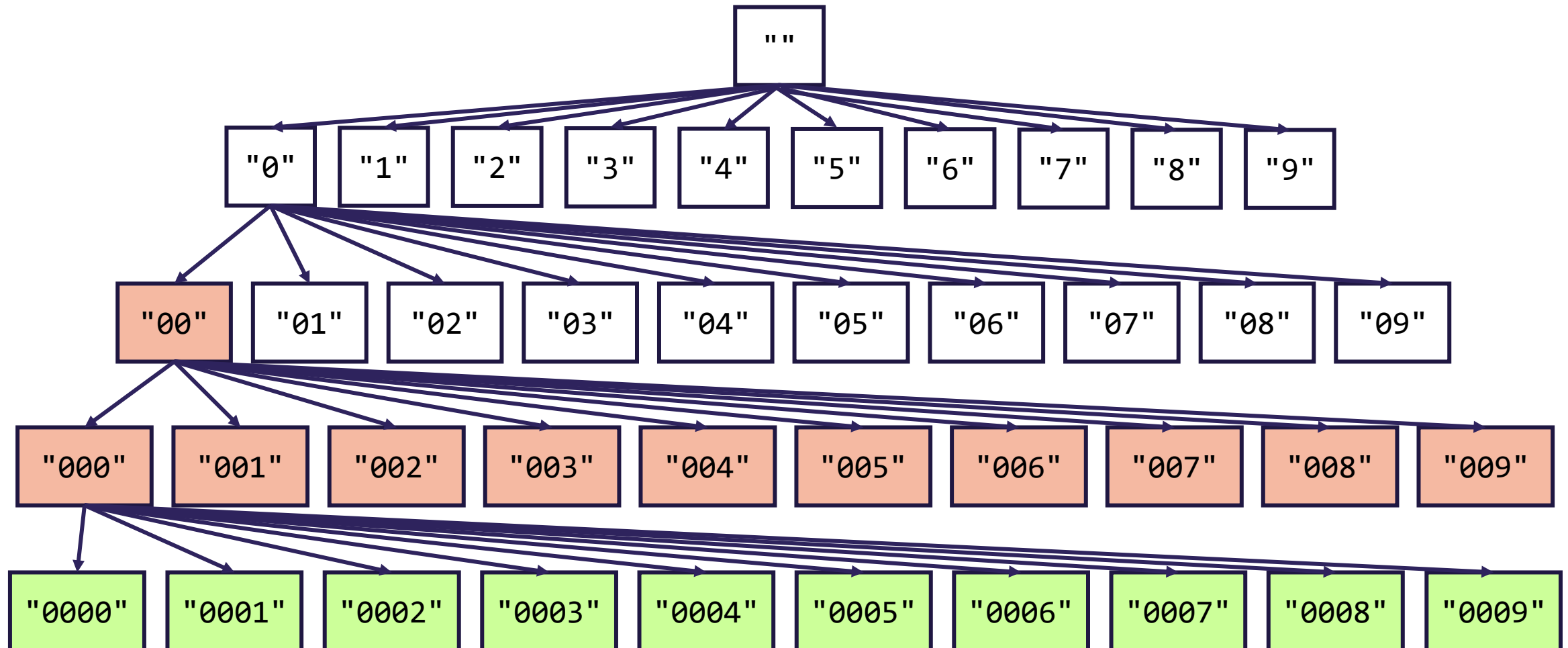
Password Cracker: End nodes will be handled by the base case

- Let's say we want to crack the password of a 4 digit combination lock



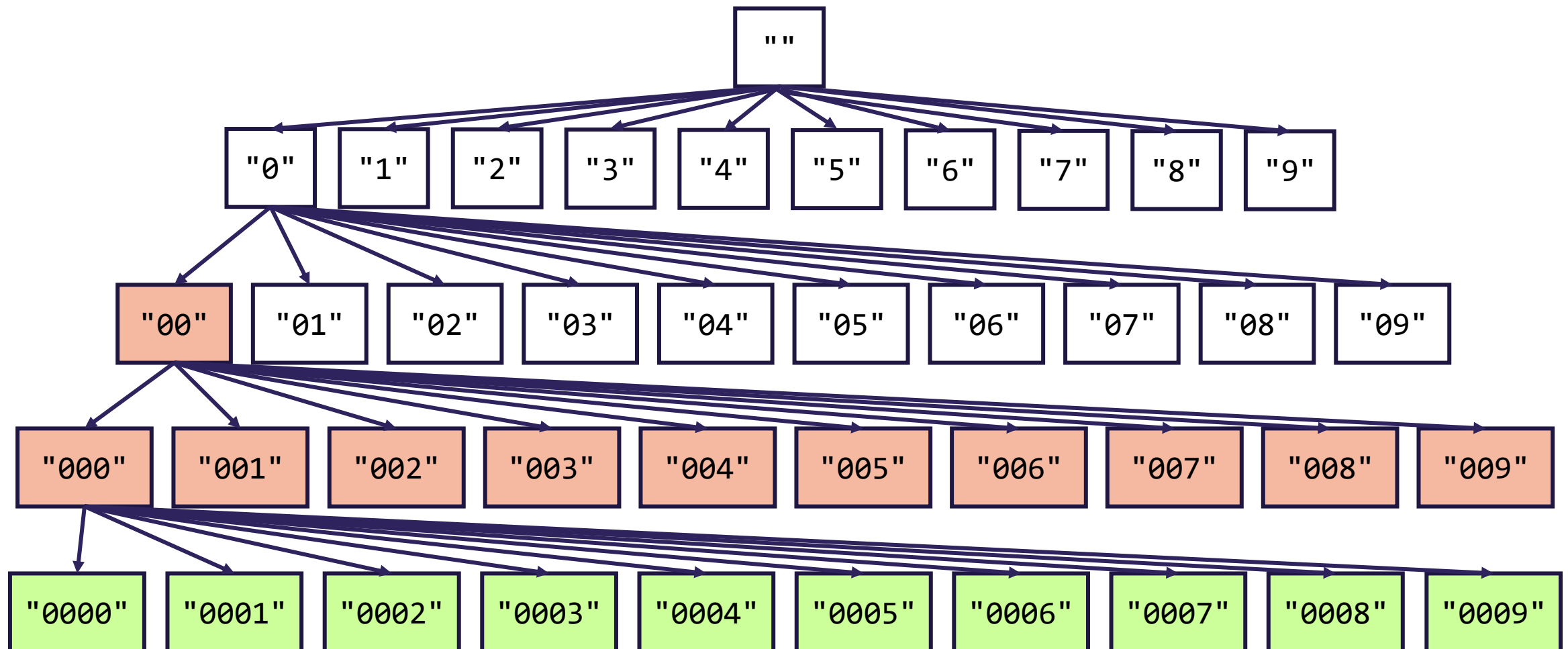
Password Cracker: Intermediate nodes are handled by the recursive case

- Let's say we want to crack the password of a 4 digit combination lock



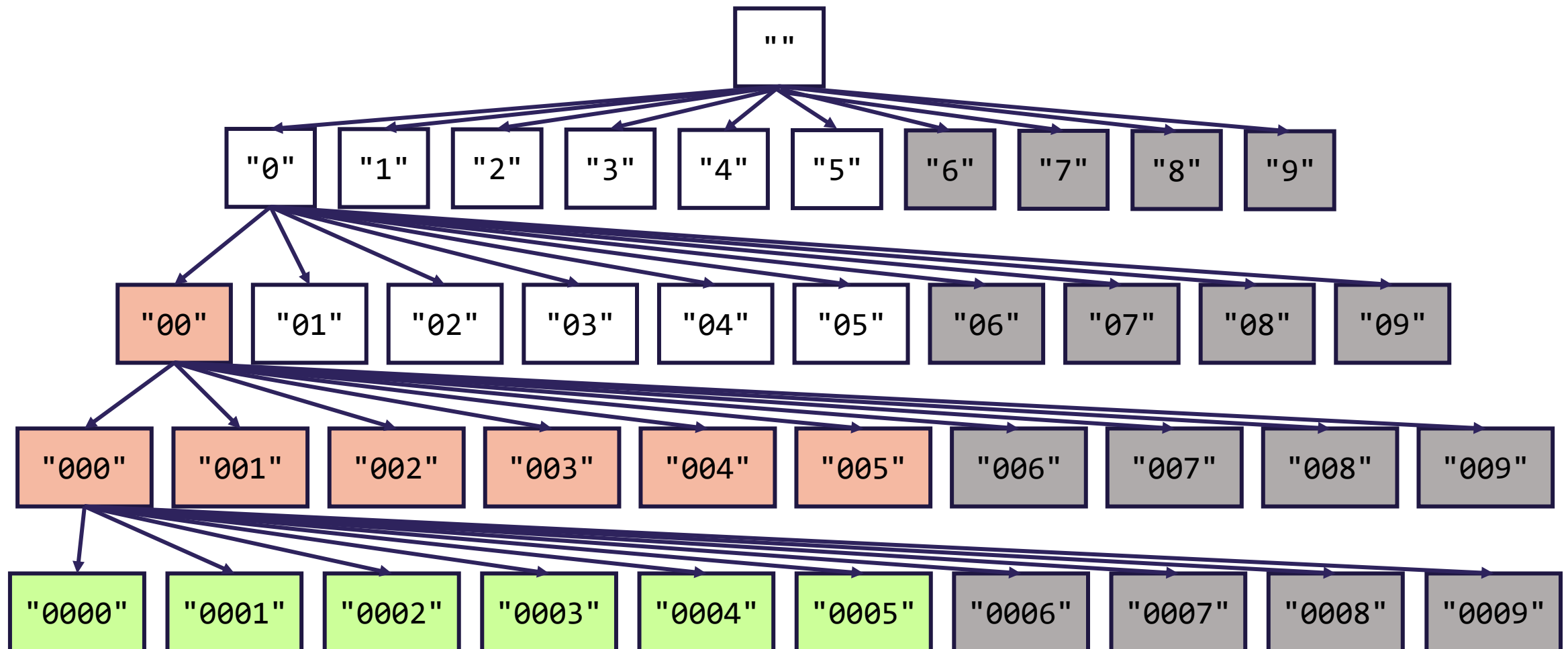
Password Cracker

- Now, what if we knew the sum of all digits was 5?



Password Cracker: Pruning Dead Ends

- Now, what if we knew the sum of all digits was 5?



Updated Exhaustive Search Pattern

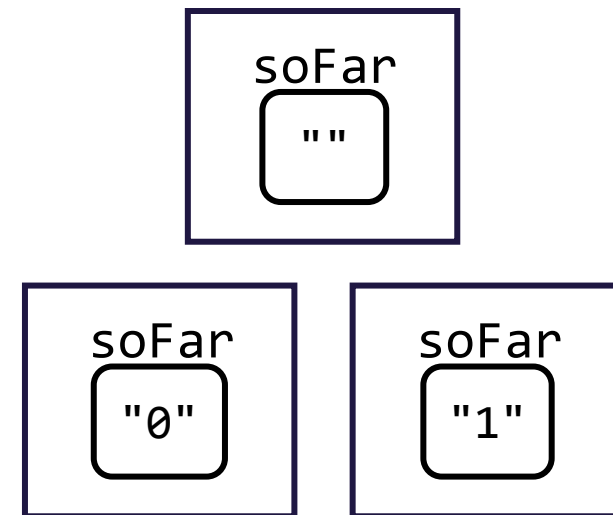
```
public static void search(input) {  
    search(input, "");  
}  
  
private static void search(input, String soFar) {  
    if (base case) {  
        // Do something with soFar (e.g. print it out)  
        System.out.println(soFar);  
    } else if (not dead end) {  
        // Might not be a loop, but 1 recursive call for each option  
        for (each option) {  
            search(input, soFar + option);  
        }  
    }  
}
```

Recursive Backtracking (Step 1)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

String soFar:

```
for (each option) {  
    search(input, soFar + option);  
}
```

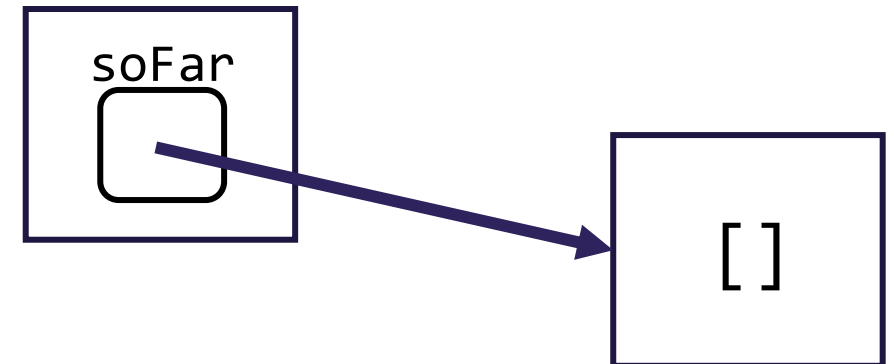


Recursive Backtracking (Step 2)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

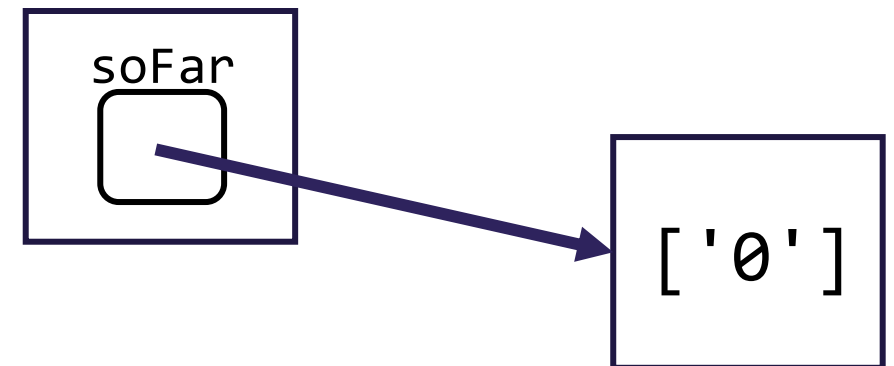


Recursive Backtracking (Step 3)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

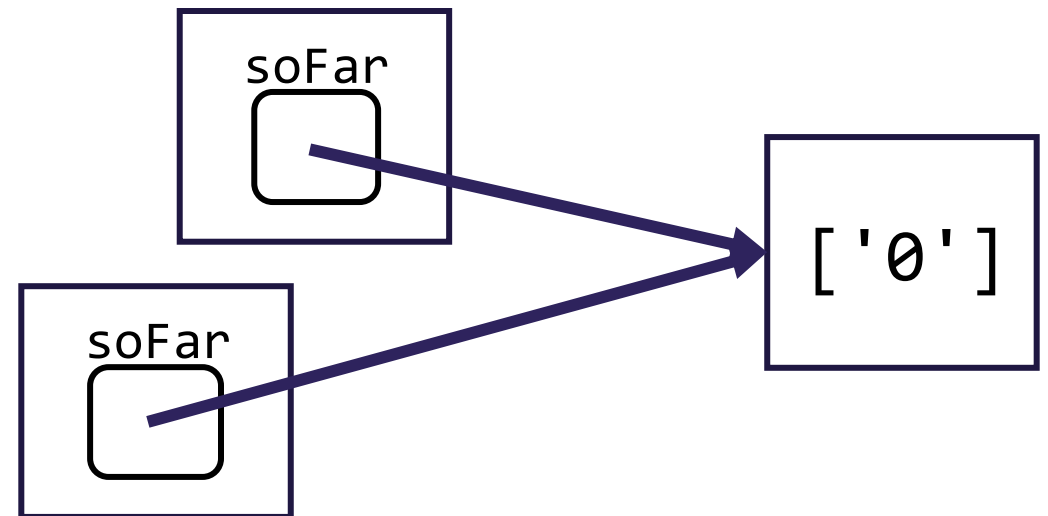


Recursive Backtracking (Step 4)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

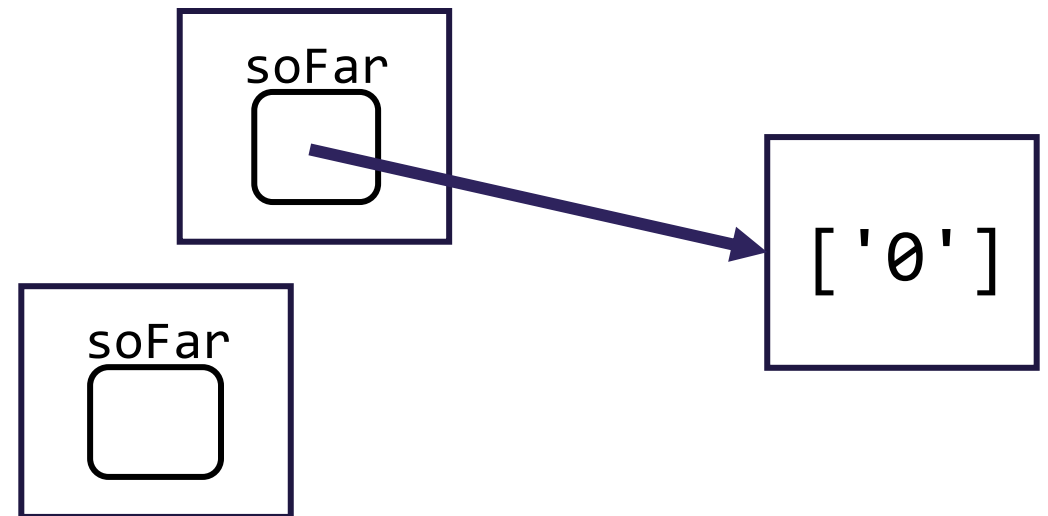


Recursive Backtracking (Step 5)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

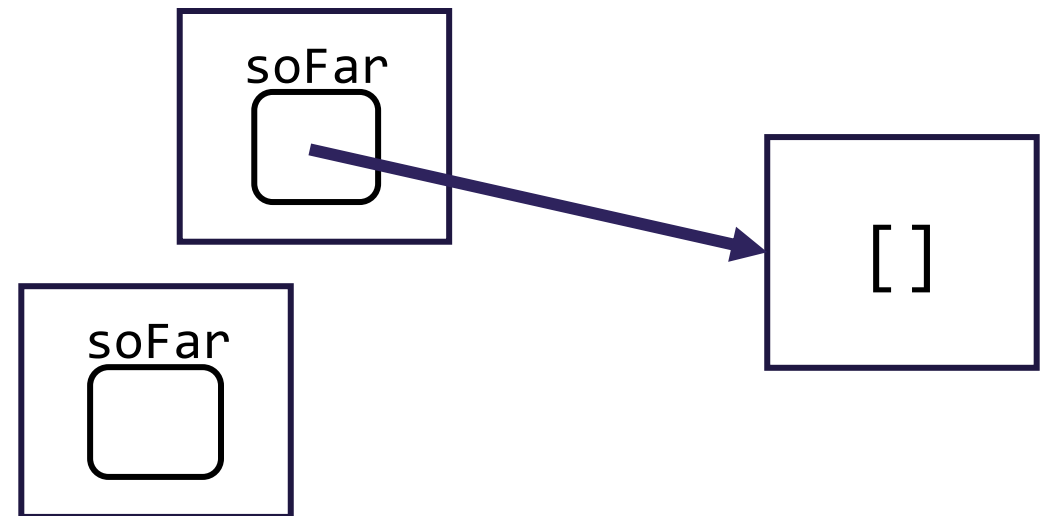


Recursive Backtracking (Step 6)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

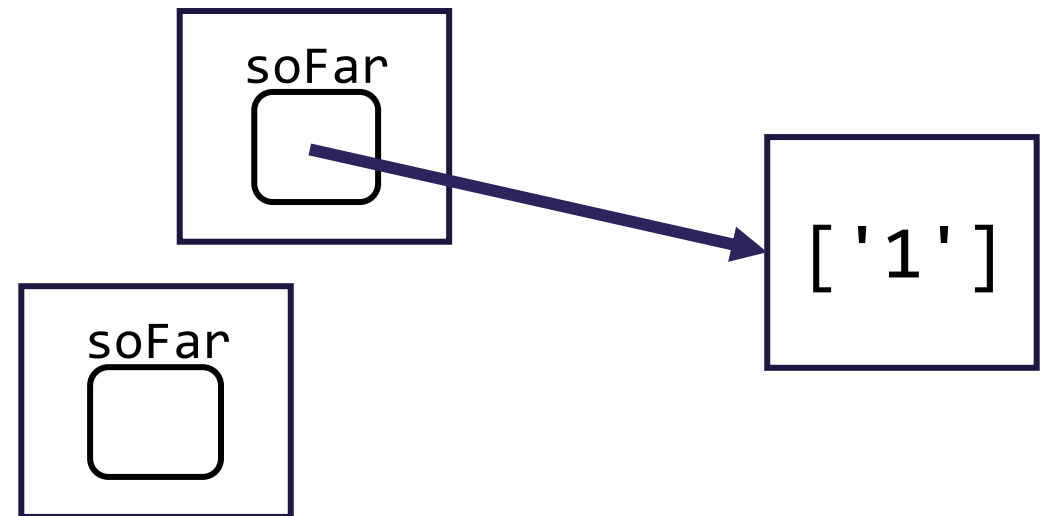


Recursive Backtracking (Step 7)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

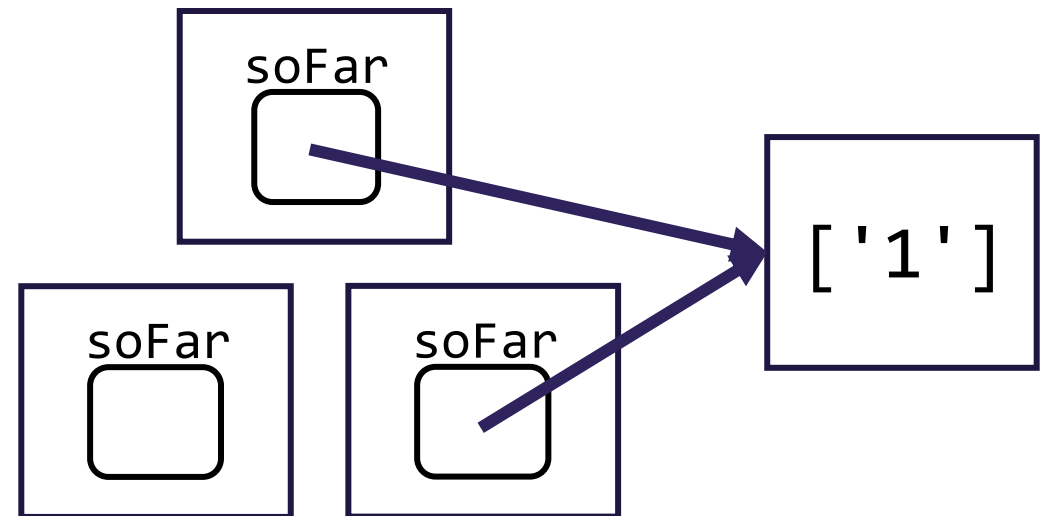


Recursive Backtracking (Step 8)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

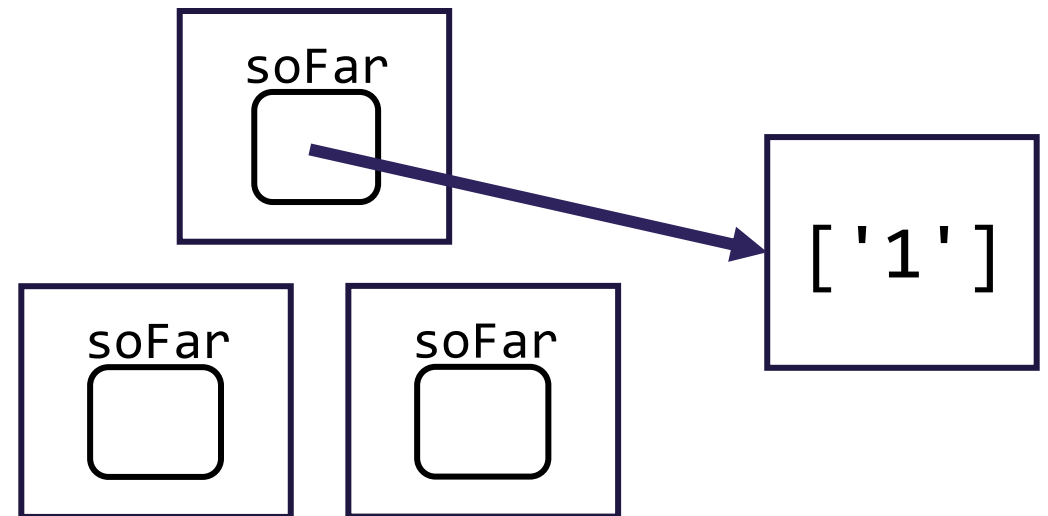


Recursive Backtracking (Step 9)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```

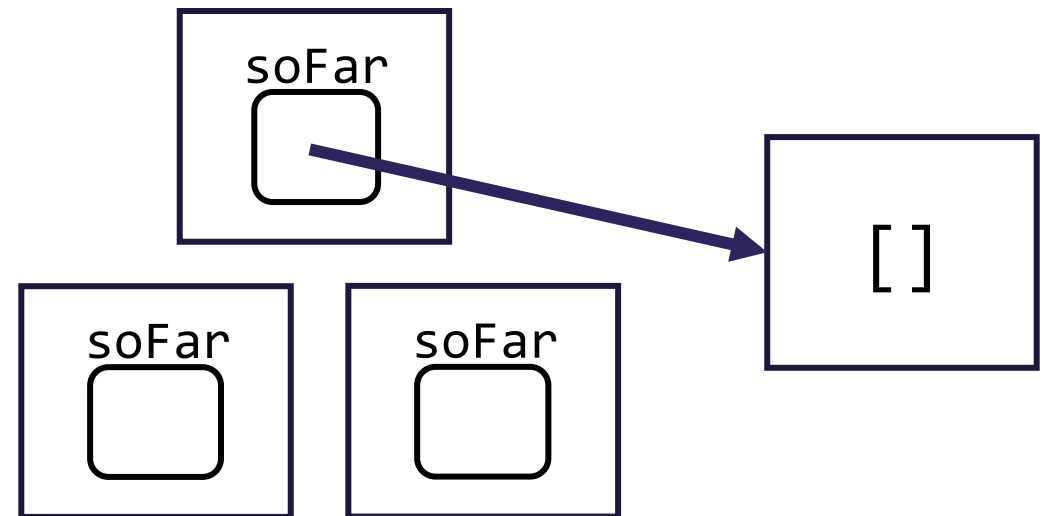


Recursive Backtracking (Step 10)

- Exhaustive search with a data structure accumulator(s)
 - Now we have to deal with reference semantics...
- Major pattern: **Choose, Explore, Un-choose**
 - All of the stack frames share the same *one* data structure
 - Need to explicitly un-choose it so it's not remembered in other frames

List<Character> soFar:

```
for (each option) {  
    soFar.add(option);  
    search(input, soFar);  
    soFar.remove(soFar.size() - 1);  
}
```



Recursive Backtracking Pattern

```
private static void search(input, List<Character> soFar) {  
    if (base case) {  
        // Do something with soFar (e.g. print it out)  
        System.out.println(soFar);  
    } else if (not dead end) {  
        // Might not be a loop, but 1 recursive call for each option  
        for (each option) {  
            soFar.add(option);                // Choose  
            search(input, soFar);              // Explore  
            soFar.remove(soFar.size() - 1);    // Unchoose  
        }  
    }  
}
```