LEC 12

CSE 123

Exhaustive Search

BEFORE WE START

Talk to your neighbors:

What's your favorite rainy day activity?

Instructors: Ziao Yin

TAs: Trien Nichole Chris Eeshani Packard

Questions during Class?

Raise hand or send here

sli.do #cse123

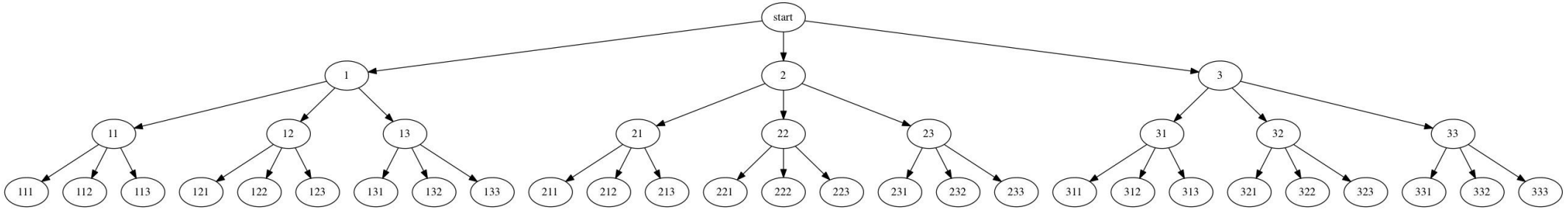
Announcements

- Creative Project 3 is out, due Wednesday, Aug 13
  - Focused on binary trees!
- Resubmission Cycle 4 closes tonight (Friday, Aug 8)
  - ***C1***, P1, P2 eligible

Exhaustive Search

- 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

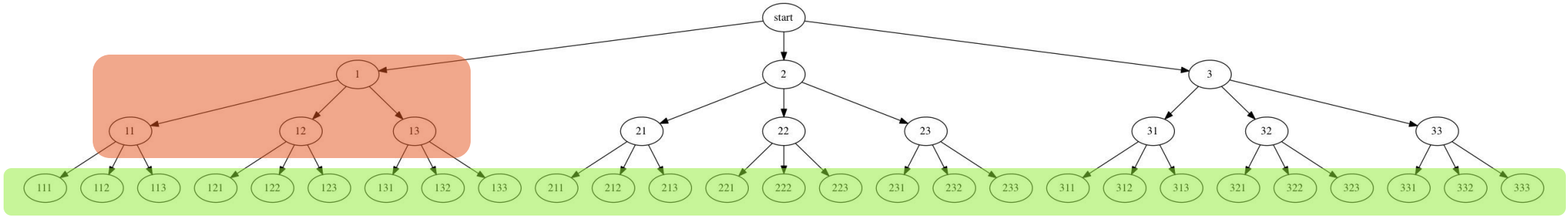
Tracing through printNums




```
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)

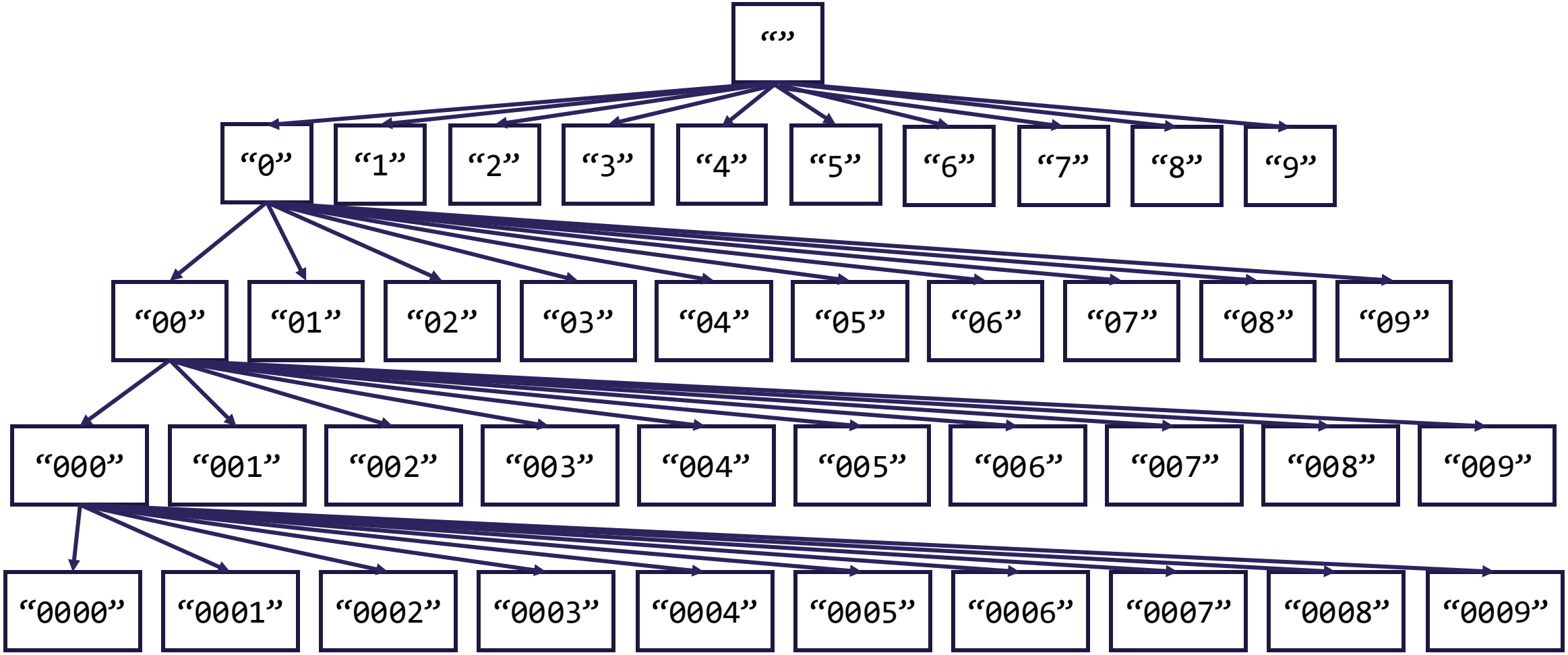
```
public static void search(input) {
    search(input, "");
}

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);
        }
    }
}
```

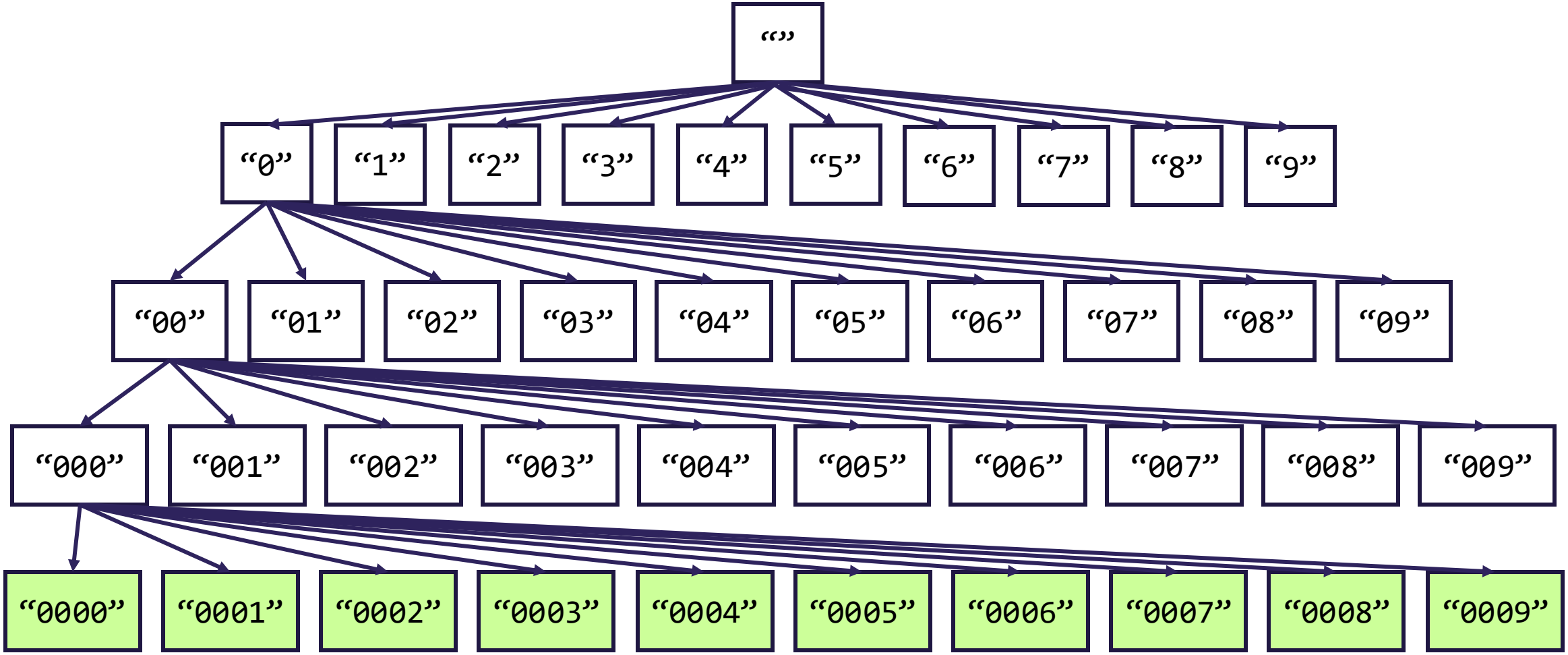
Password Cracker

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



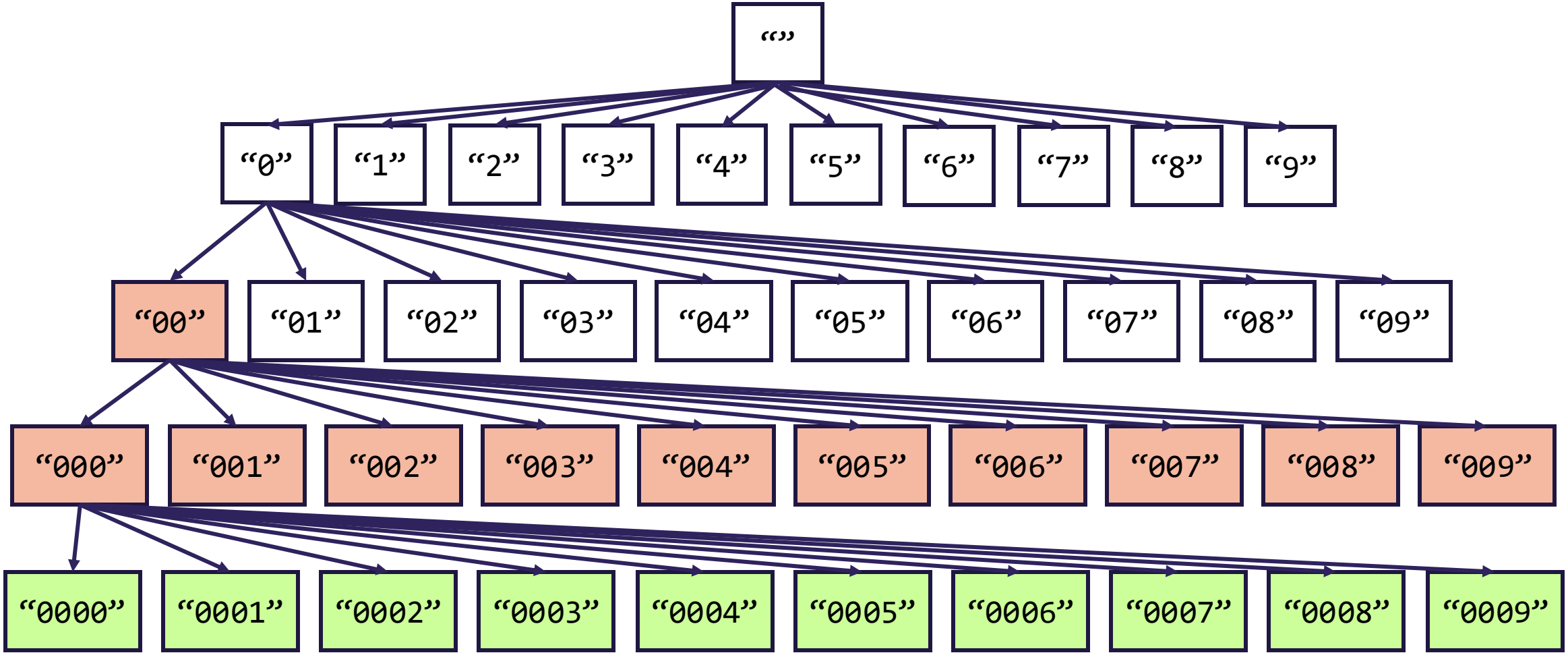
Password Cracker

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



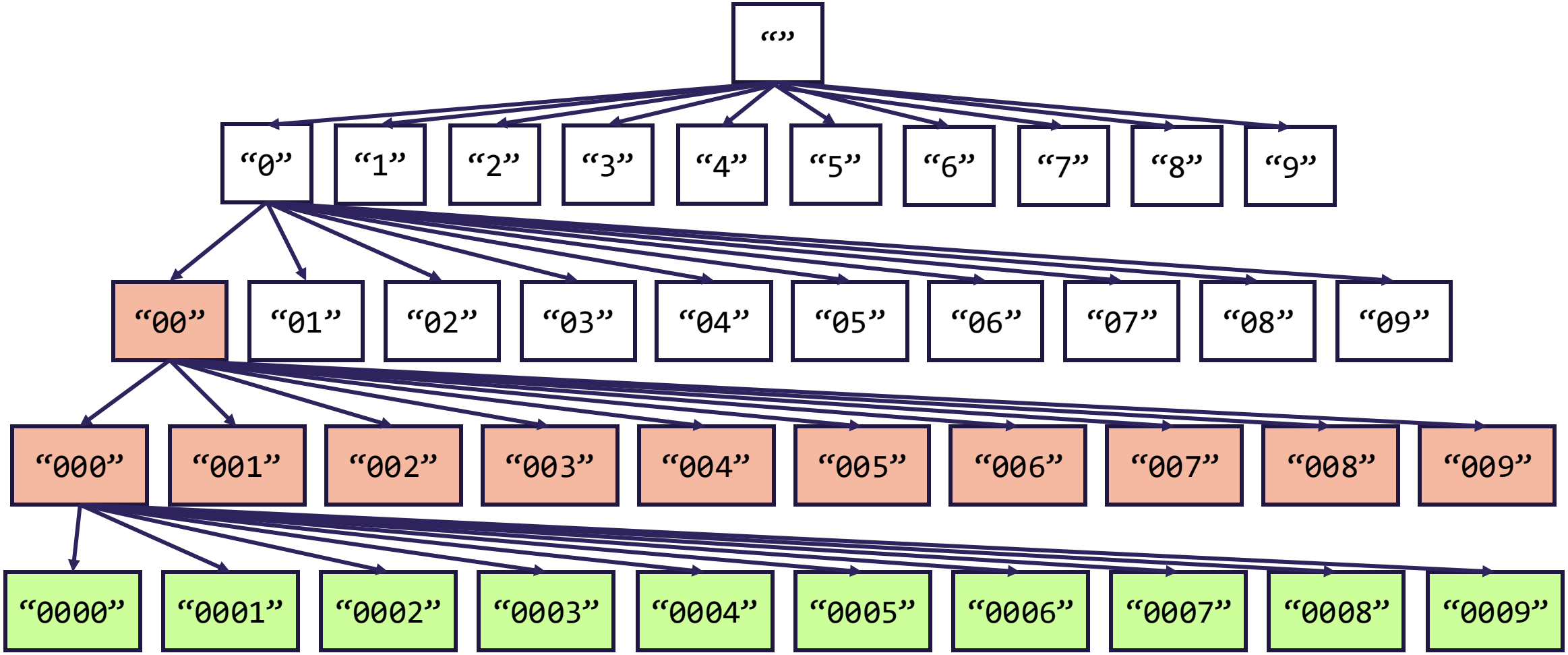
Password Cracker

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



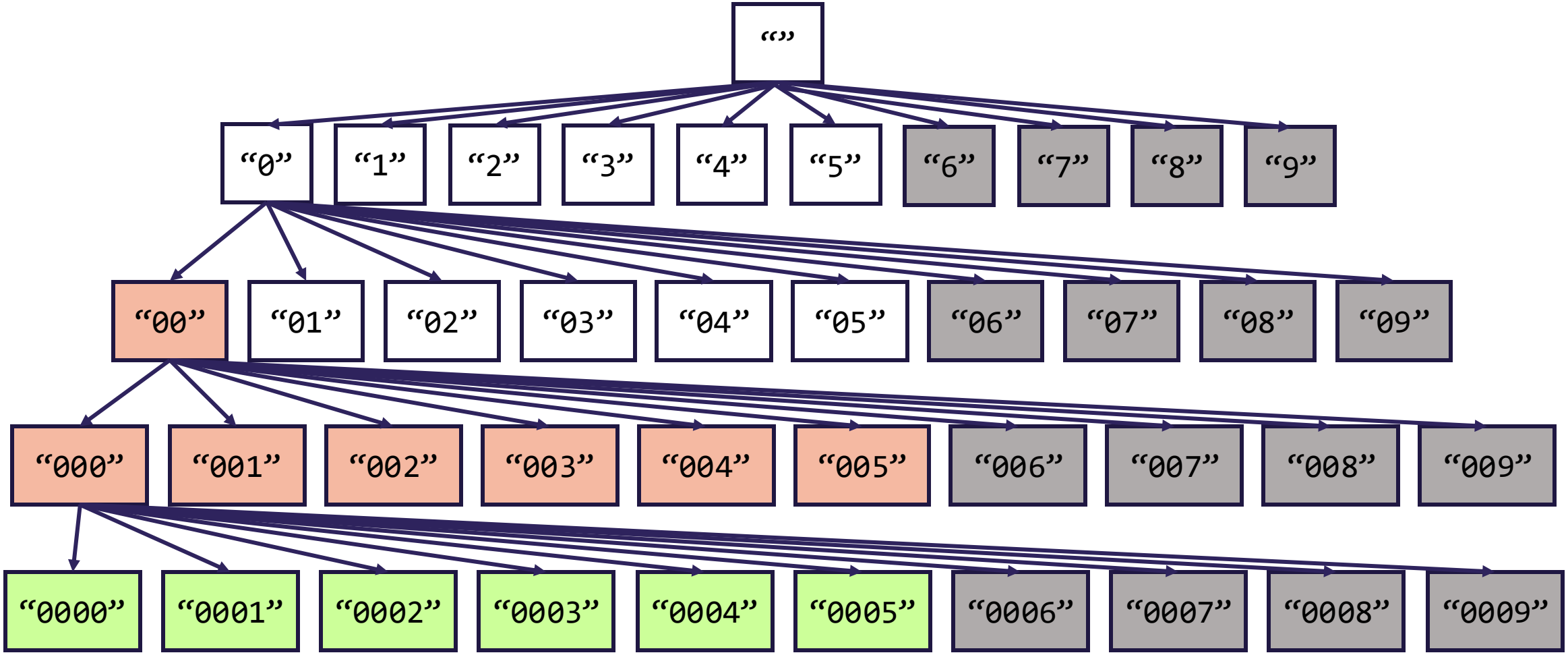
Password Cracker

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



Password Cracker

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



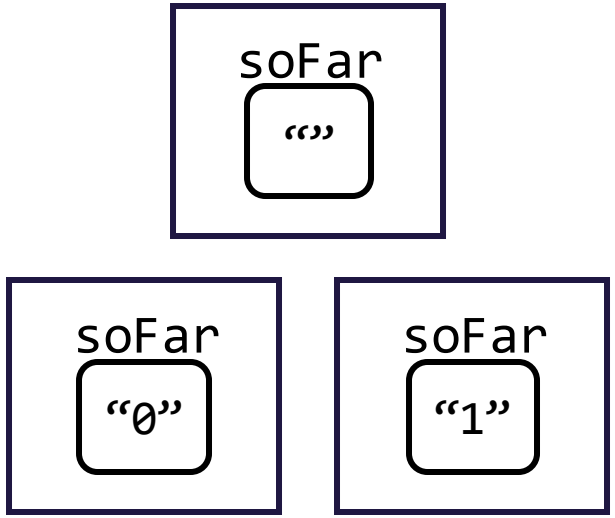
Updated Exhaustive Search Pattern

```
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

- 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);
}
```

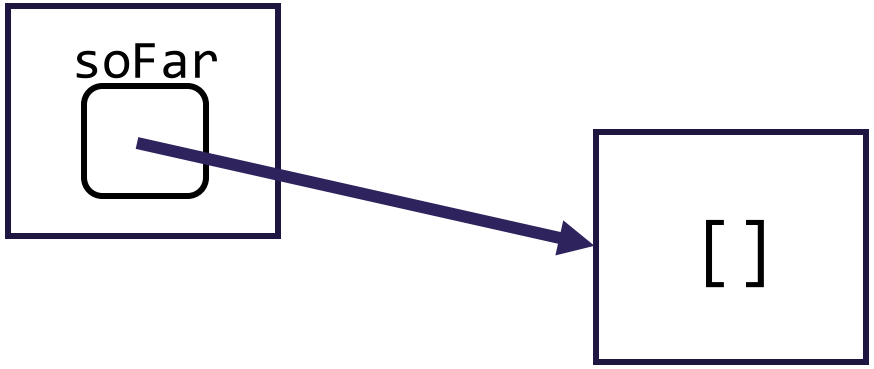


Recursive Backtracking

- 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);
}
```

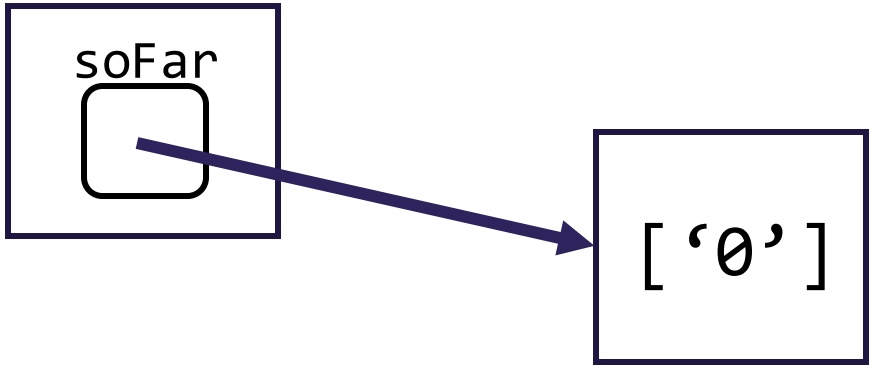


Recursive Backtracking

- 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);
}
```

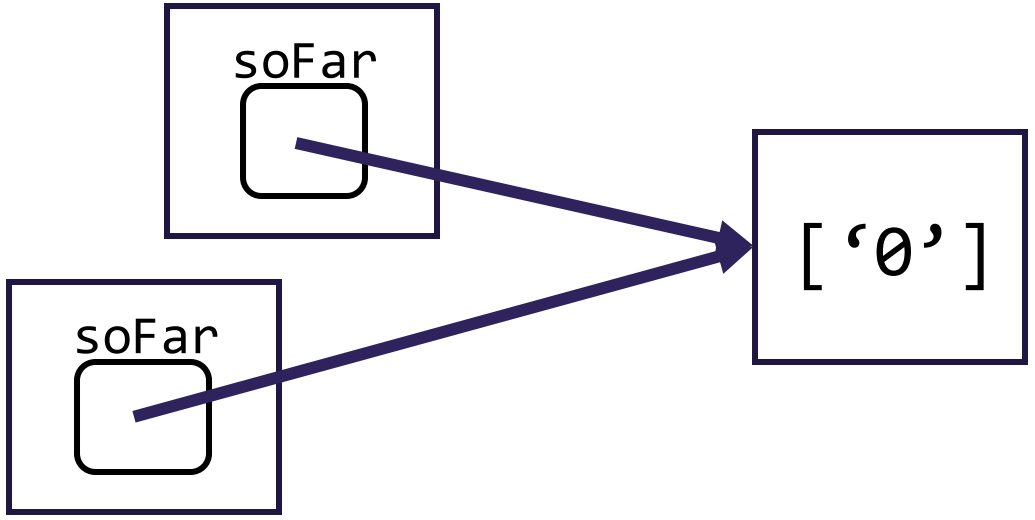


Recursive Backtracking

- 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);
}
```

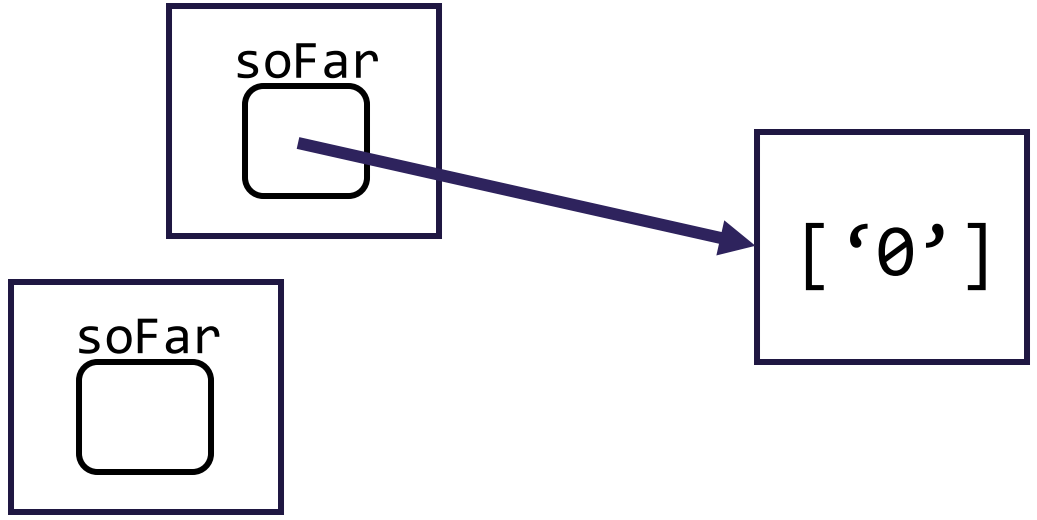


Recursive Backtracking

- 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);
}
```

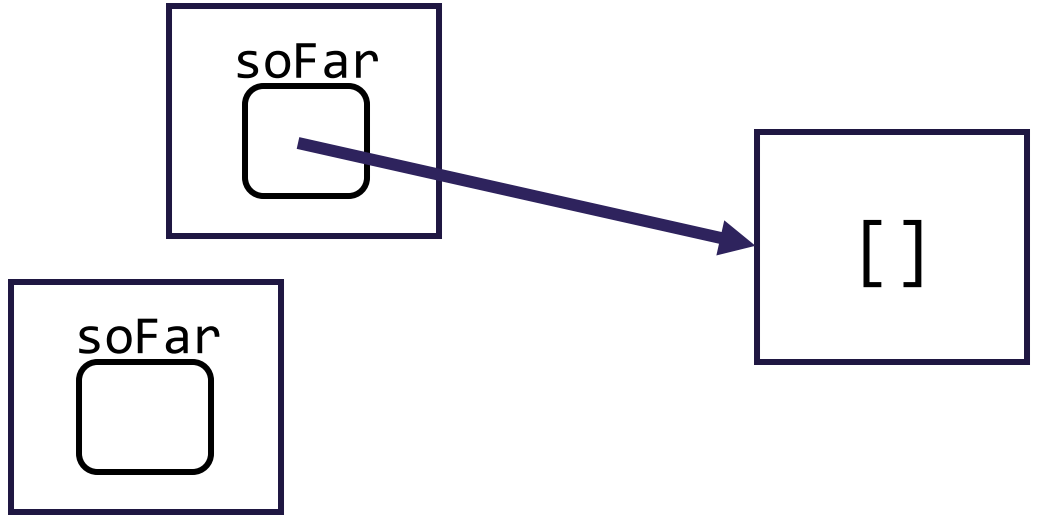


Recursive Backtracking

- 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);
}
```

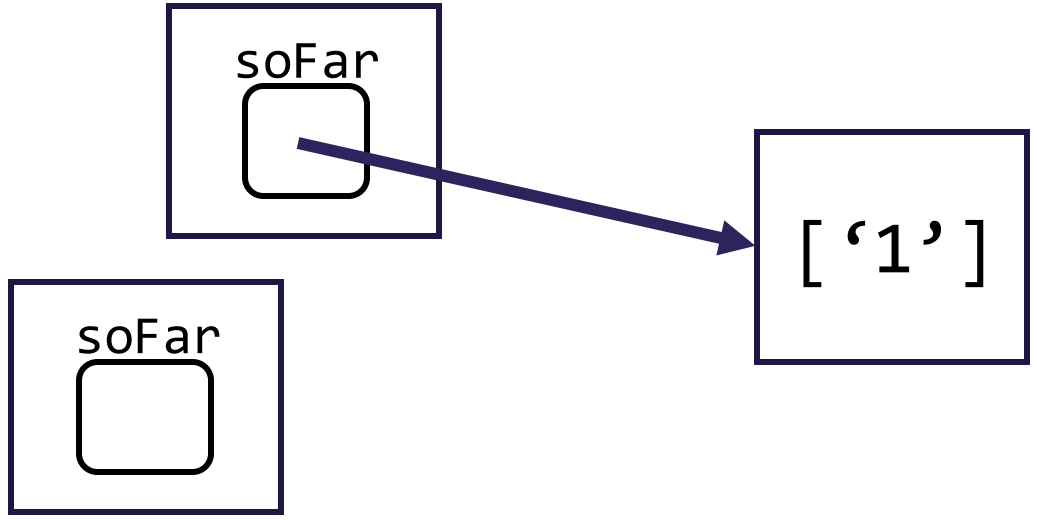


Recursive Backtracking

- 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);
}
```

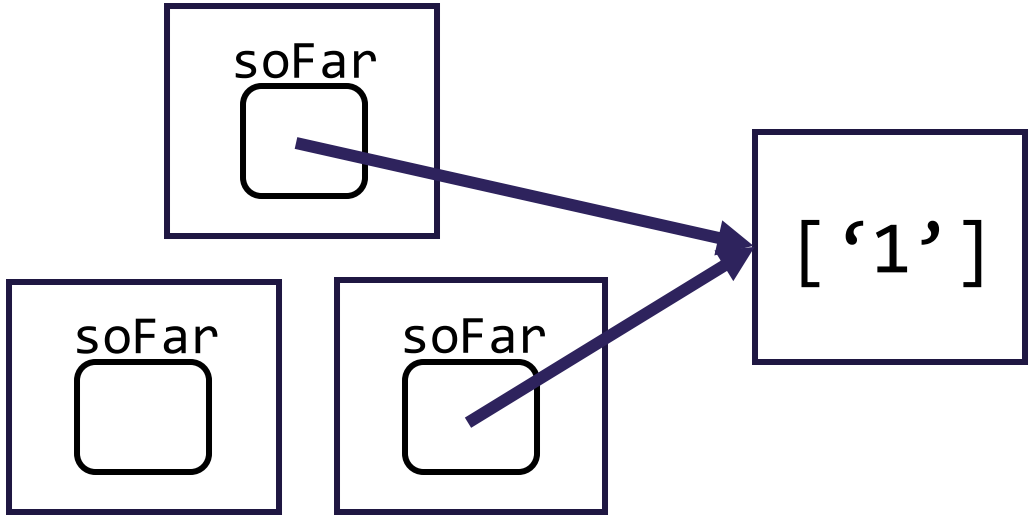


Recursive Backtracking

- 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);
}
```

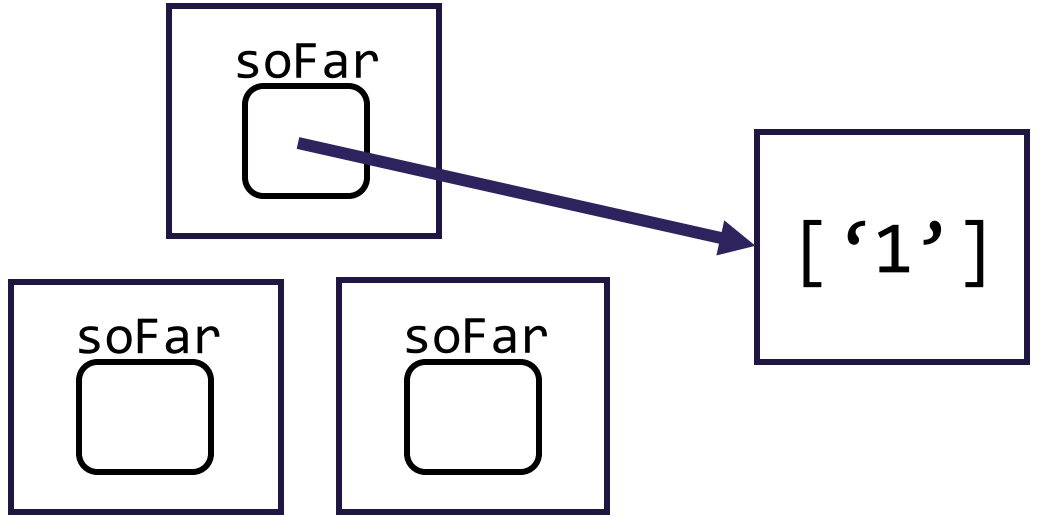


Recursive Backtracking

- 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);
}
```

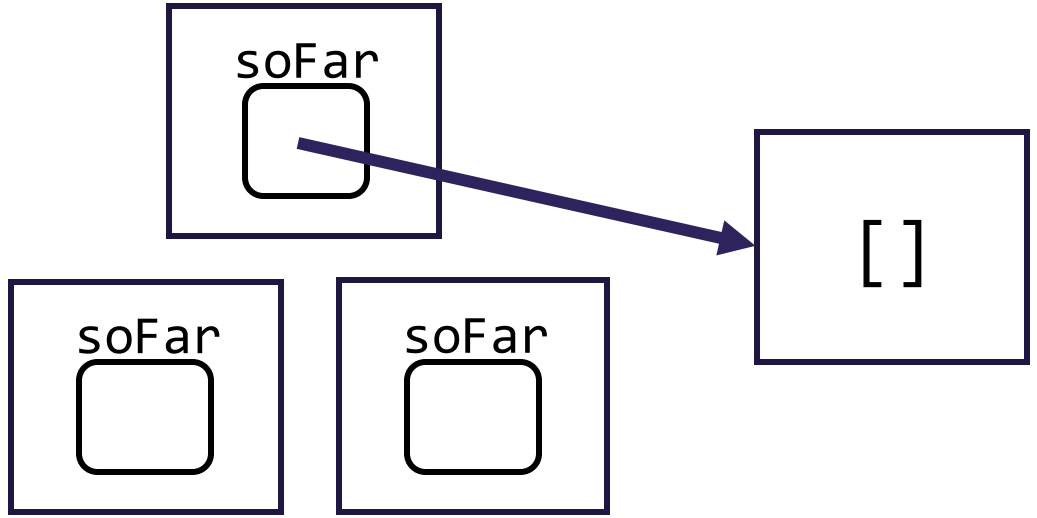


Recursive Backtracking

- 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
        }
    }
}
```