LEC 13: Exhaustive Search / Recursive Backtracking

Exhaustive Search / Recursive Backtracing

Questions during Class?
Raise hand or send here

sli.do #cse123

BEFORE WE START

Talk to your neighbors:

What’s your favorite refreshing summer drink?

Music: 123 24su Lecture Tunes ☀️

Instructor: Joe Spaniac

TAs: Andras Daniel Eric Nicole Sahej Trien Zach
Lecture Outline

• Announcements

• Exhaustive Search
  - Decision trees
  - Password Cracking
  - Dead ends

• Recursive Backtracking
  - Cipher Cracking
Announcements

• Resubmission Period 5 due tonight (8/2) at 11:59pm
• Programming Assignment 3 due Wednesday (8/7) at 11:59pm
• Resubmission Period 6 opening tonight, due next Friday (8/9)
  - Assignments available: P2, C3
• Last day of content on the final!
  - Next week: Machine learning (ML) + SpamClassifier / Hashing
  - Useful content, especially if you’re continuing to study CS
• Reminder: Grade Guarantee Calculator
  - You’ve received many, many grades throughout this quarter
  - Should have a good idea of what GPA you’re guaranteed
Lecture Outline

• Announcements

• Exhaustive Search
  - Decision trees
  - Password Cracking
  - Dead ends

• Recursive Backtracking
  - Cipher Cracking
Exhaustive Search

• Last application of recursion for the quarter!
• There are some problems computers are bad at solving
  - Polynomial vs. Nonderministic Polynomial (P vs. NP)

• Password cracking / decrypting is a great example
  - If breaking these were easy, the internet wouldn’t be useable
• So what do we do?
  - The stupid way of solving the problem
  - We “exhaustively search” through every possibility

• What do we need? Recursion + String accumulator (public / private pair)
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 {
        // Might not be a loop, but 1 recursive call for each option
        for (each option) {
            search(input, soFar + option);
        }
    }
}
Decision Trees

• Visual we use to help understand what our process is
  - Not a data structure like a Binary Tree, just a visualization tool
  - If you can make 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...)
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
Lecture Outline

• Announcements

• Exhaustive Search
  - Decision trees
  - **Password Cracking**
  - Dead ends

• Recursive Backtracking
  - Cipher Cracking
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

```java
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);
        }
    }
}
```
Lecture Outline

• Announcements

• Exhaustive Search
  - Decision trees
  - Password Cracking
  - Dead ends

• Recursive Backtracking
  - Cipher Cracking
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

```java
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

```java
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

```java
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

```java
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

```java
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

```java
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
        }
    }
}
Lecture Outline

• Announcements

• Exhaustive Search
  - Decision trees
  - Password Cracking
  - Dead ends

• Recursive Backtracking
  - Cipher Cracking