Pre-Class Work 11: Exhaustive Search

Exhaustive Search [Background Reading]

□ Motivation

Let's say we're bored one day, *extremely extremely* bored, and wanted to find out what 3-digit numbers we could make with digits 1, 2, and 3. Okay!

111, 112, 113, 121, 122, 123, 131...

Wow. That quite honestly made us more bored than when we started. But we have no choice except to **exhaustively search** every single possible option to find every single one that works. There must be an easier way to do this! We're computer scientists! We can automate anything! There's something repetitive about this, no?

For every single digit place, there can either be a 1, 2, or 3; then we have to combine that with all combinations of another digit place. How about some for-loops then? What about if they're nested?

```
for (int d1 = 1; d1 <= 3; d1++) {                   // hundreds place
    for (int d2 = 1; d2 <= 3; d2++) {               // tens place
        for (int d3 = 1; d3 <= 3; d3++) {           // ones place
            System.out.println("" + d1 + d2 + d3);
        }
    }
}
```

This looks pretty good! The inner-most loop goes through all options for the ones place while the other two digits remain the same, then once the inner-most loop finishes iterating, the middle-most loop increments, then we're printing out all combinations where the tens place is 2, and so on and so forth until the outer-most loop finishes. If this printed out, it would look like:

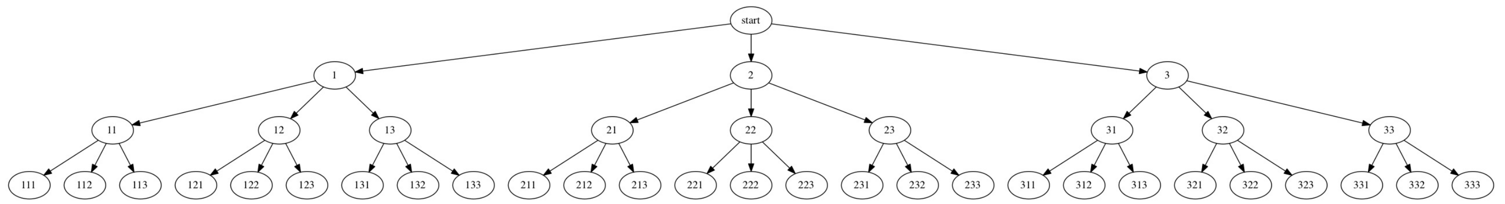
111, 112, 113, 121, 122, 123, 131, 132, 133, 211, 212, 213, 221, 222, 223, 231, 232, 233, 311, 312, 313, 321, 322, 323, 331, 332, 333

□ Decision Trees

However, the nested loops aren't super nice to look at, nor do they fully represent what's happening. What is happening? Well, we're exploring a set of choices to be made (1st digit, 2nd digit, 3rd digit).

A better way to visualize this process is with **decision trees**! It shows a series of choices or *decisions*

and the potential outcomes, and is much easier on the eyes.



Here, there will be 3 levels for each potential choice + 1 level just for the start, so 4 levels in total. Each choice has 3 possible decisions that could be made, one for each digit 1, 2, and 3. Then after that decision has been made, you have another 3 possible decisions, all the way until all 3 digits have been selected.

Therefore, there's a "branch" in the decision "tree" for each possible outcome, showing the series of choices that have been made to get there.

Each level of the tree represents a particular stage of a series of choices, with the last level showing the outcome.

Therefore, the total number of choices is the number of endpoints on the last level. With math, you would be able to calculate the total number of combinations as such: 3 * 3 * 3 = 27 (count the last level if you'd like to verify □).

Okay, you say. But what if.... We used recursion 😱 ?! Right on, let's see how that might work on the next slide!

Exhaustive search comes in handy when we're attempting to find *all* possible combinations or permutations given some type of limitation or conditions. It's useful for generating possibilities and various outcomes.

□ Main Points

- We wanted to find out all 3-digit numbers we could make with 1, 2, and 3.
- We started out by trying to do this by hand and **exhaustively searching**, but then we developed an algorithm that involved nested loops to do this for us.
- While this worked, nested loops were not super fun for us to look at, so we decided to visualize this process with **decision trees** that show a series of choices or *decisions* and the potential outcomes.
- **Exhaustive search** is useful when we want to find *all* possible outcomes given some limitation or conditions.

Anatomy of Exhaustive Search [Background Reading]

How would **exhaustive search** be implemented recursively? Well, as we saw with recursion, we have lots of cases to break down. Firstly, what would our base case be? What would be our general case? And how would we be able to keep track of all of the combinations we've already tried?

We'll break down the code in a bit, but this is how it would be implemented:

```
public static int count = 0;
public static void main(String[] args) {
    printNums();
    System.out.println(count);
}

// Prints out all 3-digit combinations of
// 1, 2, and 3
public static void printNums() {
    printNums("");
}

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

As with many recursive methods, there is a `public` and `private` pair. The public method calls its recursive private partner method and in the private method, there's the base case and a recursive case. The base case is when the string `soFar` is of length 3, meaning that a 3-digit combination is ready to be printed! The recursive case is when there are still digits left to be chosen for a combination.

soFar

Before we break down the recursive case, let's go over what `soFar` is. From what we know about scope, after a for loop ends or a method ends, whatever variables or data that was created within those curly braces will get garbage collected by Java. But we need to remember what choices have already been made, or in this case, which number we've already chosen for the other digits.

How will we manage that, if everything we create to remember what we've chosen is gone by the

time we try a different combination?

Note that whenever we pass a primitive data type variable or a String, it is passed-by-value and simply a copy of the data. In this case, an empty String called `soFar` is being used to keep track of the combinations, where every single new digit that we try is then appended to `soFar`.

🔍 Breaking Down the Recursive Case

Now, let's break down what is going on in the recursive case by going through what happens when it runs for the first time.

The very first recursive call to `printNums` has `soFar = "1"`, then goes into the else case as `soFar.length() != 3`. It then calls `printNums` again, however, `soFar` is now `"11"`. It once again goes to the else case then calls `printNums` with `soFar = "111"`. Finally, we hit the base case and `"111"` gets printed out. Then we go all the way back to when we called `printNums` with `soFar = "11"`, and move onto the next line where `printNums` is called with `soFar = "112"`.

This occurs again and again, until all of the combinations are printed out and we finish executing the very first `printNums` that we called.

👀 Preview: **Choose-explore-unchoose**

In the following lesson, we'll introduce a recursion pattern called **choose-explore-unchoose**. Let's see how it can be used for **exhaustive search**!

```
public static void main(String[] args) {
    printNumsCEU();
    System.out.println(count);
}

// Prints out all 3-digit combinations of
// 1, 2, and 3 using the
// choose-explore-unchoose pattern
public static void printNumsCEU() {
    printNumsCEU(new ArrayList<Integer>());
}

private static void printNumsCEU(List<Integer> chosen) {
    if (chosen.size() == 3) {
        count++;
        System.out.println(chosen);
    } else {
        for (int i = 1; i <= 3; i++) {
            chosen.add(i);
            printNumsCEU(chosen);
```

```
            chosen.remove(chosen.size() - 1);
        }
    }
}
```

Everything has a similar structure, however now the recursive case has changed, and instead of a String `soFar` keeping track of the past combinations, there's a list called `chosen`.

Before we break down the recursive case, let's go over what `chosen` is. Again recall that we need some type of way to keep track of the current combination. Remember that with `soFar`, as it was a String, we were passing around a copy.

Here, we instead pass around a reference to `chosen`, which gets updated within the recursive case. But what is going on in the recursive case?

Choose-explore-unchoose Recursive Case

Let's break it down. The for-loop explores all possible options for each digit in the 3-digit combination (1, 2, or 3). Within the loop, we *choose* the digit we'll be trying by adding it to `chosen` so that we remember the current digits in the combination that we're trying. Then we call `tryCombination` to try the other digits in the combination.

Through recursion, it goes through all the other digits left in the combinations to *explore* other options. Then at the very end, we need to reset the combination to retry another one! Back when we had `soFar`, we simply reverted back to a version of `soFar` that did *not* have the new digit appended.

But with chosen, we need to manually revert it. Thus, we *unchoose* the most recently added digit from `chosen`, and the loop goes to another number.

Woah. That was a lot. Don't worry. This **choose-explore-unchoose** pattern will be explored more in the next lesson of **recursive backtracking**.

Main Points

- In **exhaustive search**, the **base case** represents the simplest case that stops **recursion**.
  - For example, in our search for 3-digit combinations, the base case occurs when we have a combination of length 3.
- The **recursive case** is when the method calls itself with arguments we modified so we can progress towards the **base case**.
  - We want to consider all possible choices for the next element in the number combination.
- We want to use a variable to keep track of the choices we made during exploration so that the recursive function to remember and update the current state of the problem.

    - In `printNums`, the tracker variable was `soFar`; in `printNumsCEU`, the tracker variable was `chosen`
- The **choose-explore-unchoose** pattern involves choosing an option, exploring it recursively, and then unchoosing it to explore other options so we can effectively manage our combinations!

Exhaustive Search Example

This code slide does not have a description.

printNumbers

Write a recursive method `printNumbers` that prints all of the integers that are composed of two 2's and two 5's on separate lines. One example output to `printNumbers()` is shown below.

```
2255
2525
2552
5225
5252
5522
```

Below is the decision tree: