[In Class] Lesson 11 (11/6)

Review: Permutations

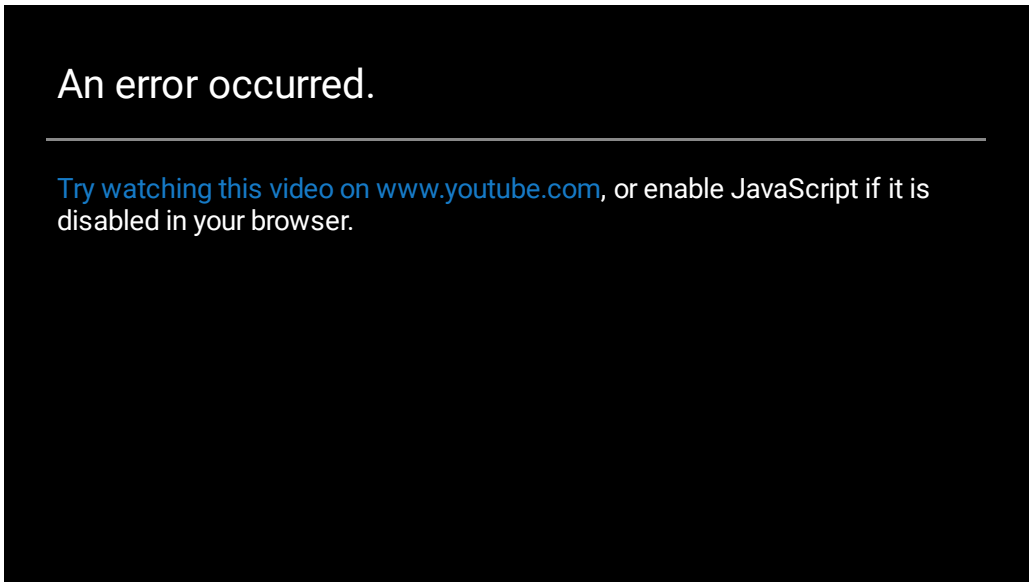
In the pre class work you were shown some examples of how to use **exhaustive search** to produce all possible combinations, or "permutations", of the three digits 1, 2 and 3.

How can you extend this code to produce all combinations of 1, 2, 3 and 4?

How can you extend this code to be able to produce all combinations of digits 1 through 9 based on a given int parameter?

Pick the lock

Assume for a moment, you are a dishonest man.



Or perhaps assume you are the type of person who commonly forgets the combination to your padlock. The only way to get a lock open to which you do not have the combination would be to try every single possible combination until you find the right one.

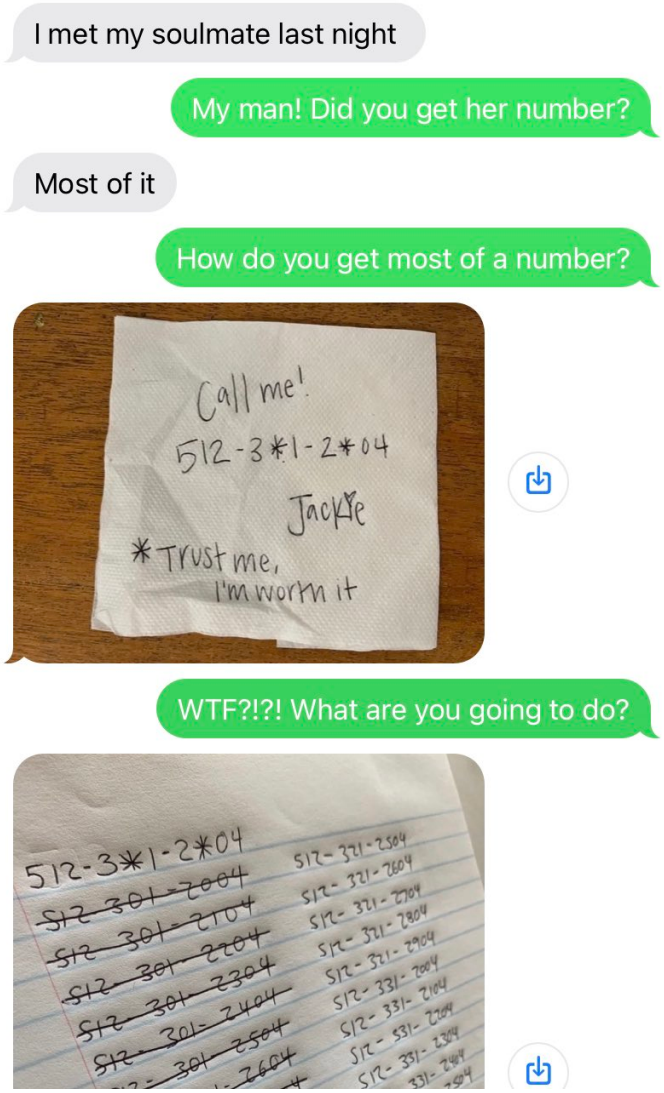
This lock requires a combination of 4 digits, each of which may be the value of 0 - 9. Using exhaustive search and recursion write the code to produce all possible combinations to try.

Extension: How much you change your code to pick this lock instead? This lock's combination requires 3 numbers, each of which can range from 0 - 39.


HARDENED
MASTER
35
0
5
10
15
20
Master

Smudged Phone Number

It's Friday night and you spot a cutie across the room at a party. You just learned about recursion in CSE 123 so you're feeling extra smart and capable, so you walk over and strike up a conversation. You two hit it off and you work up the courage to ask for their number. They grab a napkin, scribble something down and hand it to you. You're so excited you grab the napkin and quickly stuff it into your pocket. You leave the party feeling on top of the world- until Saturday morning you wake up and look at the napkin. You realize in your haste to make a smooth exit you smudged a couple of the numbers your crush wrote down and now you can't read them. What do you do?



^ according to the internet this is a real thing that happened, shout out to Jackie for giving us a good exhaustive search problem!

Write a program that will produce all possible combinations of a phone number that is missing 2 numbers.

Guess the password

The challenge here seems simple: you need to guess a 4-bit password.

The problem is that you know the system might lock you out after some number of incorrect attempts! Thankfully, you managed to watch someone keying in the password from a distance, so you have a pretty good guess as to what it is.

You start with a best-guess at what the password is, and a *confidence rating* between 0 and 1 for each of the bits. A higher confidence rating means you are more confident in the bit at this position.
For example:

```
boolean[] passwordGuess =    {false, true,  true,  false, false, true,  false, true};
double[] confidenceRatings = {0.95,  0.76,  0.62,  0.92,  0.64,  0.68,  0.52,  0.93};
```

Would mean that you are *most* confident in the 0th bit (you are very sure it was `0`!) and you are *least* confident in 7th bit (you think it was probably a `0` but it could definitely have been a `1`!)

Goal:

Generate every possible password, in the order of most likely guess to least likely guess. Use a recursive strategy!

Hint: the most likely password is just... `passwordGuess` :)

A good starting point would be to consider what the first, second, and third guesses should be. Then think about how your recursive strategy should work.

Background:

This is a simplified version of an actual problem that cropped up in research done here at UW! In the real scenario we were not guessing a password, but a cryptographic key that was 128 bits long. We *did* actually have confidence ratings for our guesses of each bit, but they were gained from measuring tiny differences in electrical behavior of a circuit, not from looking over someone's shoulder.

Extra challenges (completely optional, fun extensions):

Because the key was so long, checking every single one was impossible (there would be 2^128 of them!) so we determined a maximum number of bits we would ever try the inverse of (say, 3). If you

are interested, consider how you might rewrite you solution to try every possible password, but only ever allowing 2 bits to be changed, still in order of confidence!

You'll note that all of the confidence ratings are `>= 0.5`. Why is that?

Another real challenge we had was that our method for guessing bits had a 50% chance to *always* be wrong for every bit. (That is, if it guessed 101110, the right key would be 010001). How would you modify your search to generate these guesses as well, where you consider the inverse of each guess to be just as likely as that guess?

If the correct key is our last guess, it is easy to calculate how many tries we will make: 2^n where n is the length of the password/key. (So for our 4-bit password, we'd try all 2^4 possibilities.) Trickier is computing how many guesses it will take for some password in the middle. Consider how to compute the number of tries your implementation will make for a given password (e.g. pick one at random, and figure out how many recursive calls will happen!)