Outline

1. NQueens
2. Sentence Splitter

NQueens Problem

The NQueens problem is the challenge to place \( n \) queens on a chess board so that none of them are attacking each other.

We will begin by solving this problem using for loops, and then we will solve it much more elegantly using recursive backtracking.

Implementing a Tiny Piece of Google

When you enter a query with no spaces like `thisisasentence` into Google:

It fixes it into **this is a sentence** using recursive backtracking.

Sentence Splitting

Given an input string, \( \text{sentence} \), containing no spaces, write a method:

```java
public static String splitSentence(String sentence)
```

that returns \( \text{sentence} \) split up into words.
Sentence Splitting

Given an input string, `sentence`, containing no spaces, write a method:

```java
public static String splitSentence(String sentence)
```

that returns `sentence` split up into words.

To do recursive backtracking, we need to answer these questions:

- What are the choices we’re making incrementally?
  - which character to split at
- How do we “undo” a choice?
  - re-combine a string by the char we split at
- What are the base case(s)?
  - our left choice isn’t a word and our right choice IS a word

It helps to answer these questions for a particular input. So, pretend we’re working with:

```java
thisisasentence
```

Sentence Splitter Solution

```java
1 public String splitSentence(String sentence) {
2   // The entire sentence is a dictionary word!
3   if (words.contains(sentence)) {
4     return sentence;
5   }
6   // Try splitting at every character until we find one that works...
7   for (int i = sentence.length() - 1; i > 0; i--){
8     String left = sentence.substring(0, i);
9     String right = sentence.substring(i, sentence.length());
10    // If the left isn’t a word, don’t bother recursing.
11    // If it is, split the remainder of the sentence recursively.
12    if (words.contains(left)) {
13      right = splitSentence(right);
14      // Since the left was a word, if the right is also an answer,
15      // then we found an answer to the whole thing!
16      if (right != null) {
17        return left + " " + right;
18      }
19    } // Undo our choice by going back to sentence
20  }
21  return null;
22 }
23 ```

One More Important Choice

When doing recursive backtracking, we need to differentiate between:

- finding a result
- failing to find a result (e.g., backtracking)

Generally, we do this by treating null as a failure. For example:

- On the input, “thisiasentence”, none of the recursive calls should return “thisis”, because it isn’t a word.
- If we get down to an empty string, that would indicate a failure; so, we’d return null