Recursive Backtracking

COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES
YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.

MAP FOR X:
1  Sentence Splitter

2  Playing With Boolean Expressions
Recursive Backtracking

**Definition (Recursive Backtracking)**

**Recursive Backtracking** is an attempt to find solution(s) by building up partial solutions and abandoning them if they don’t work.

**Recursive Backtracking Strategy**

- If we found a solution, stop looking (e.g. return)
- Otherwise for each possible choice $c$...
  - Make the choice $c$
  - Recursively continue to make choices
  - Un-make the choice $c$ (if we got back here, it means we need to continue looking)
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, `sentence`, containing **no spaces**, write a method:

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

that returns `sentence` split up into words.
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?
- How do we “undo” a choice?
- What are the base case(s)?

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

```
thisisasentence
```
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
```
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, "**thisisasentence**", 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`
public String splitSentence(String sentence) {
    // The entire sentence is a dictionary word!
    if (words.contains(sentence)) {
        return sentence;
    }

    // Try splitting at every character until we find one that works...
    for (int i = sentence.length() - 1; i > 0; i--){
        String left = sentence.substring(0, i);
        String right = sentence.substring(i, sentence.length());

        // If the left isn’t a word, don’t bother recursing.
        // If it is, split the remainder of the sentence recursively.
        if (words.contains(left)) {
            right = splitSentence(right);
            // Since the left was a word, if the right is also an answer,
            // then we found an answer to the whole thing!
            if (right != null) {
                return left + " " + right;
            }
        }

        // Undo our choice by going back to sentence
    }

    return null;
}
You may have noticed that many of the class examples I've been showing involve me using a class that I've already written. There are several reasons for this:

- Learning to read and use an API is a really important programming skill
- Switching between the client and implementor views is an important goal of this course
- The code I write is usually easy, but really tedious (so, it would be a waste of time to write in class)

Take-Away

Every time I print out an API for you, you should try to understand it from the comments. This will help you on the homework, on exams, and in any future programming endeavors.
Today's API is BooleanExpression.

What is a BooleanExpression?

The BooleanExpression class allows us to represent the conditions we write in if statements. For instance, to represent the following:

```java
if (!(queue.size() > 0) && queue.peek() > 5) {
    ...
}
```

We would do

```java
new BooleanExpression("(!a && b)");
```

Notice that we use single letter variable names instead of `queue.size() > 0`. This is a simplification for implementation.
Evaluating Boolean Expressions

Remember when we took \((1 + 2) \times 3\) and evaluated it to 9 recursively? We can do a similar thing for Boolean Expressions:

Consider the Boolean Expression from above:

\[
(!a \land b)
\]

Suppose we know the following:

- a is true.
- b is false.

What does this expression evaluate to?

\[
(!a \land b) \rightarrow (!true \land false) \rightarrow (false \land false) \rightarrow false
\]

Suppose we wanted to write a method:

```java
public static boolean evaluate(BooleanExpression e, ??? assn)
```

where `assn` represents the truth values of the variables.

What type would `assn` be? It's a **mapping** from variables to truth values.
Okay, so, we have:

```java
public static boolean evaluate(BooleanExpression e, Map<String, Boolean> assignments)
```

Consider the following case:

- `e` is `a && b`
- `assignments` map is `{a=true}`.

What should `evaluate` return?

We can’t answer the question. What seems like a good idea? `null`.

So, we change the return type to `Boolean`.
Who Should Implement `evaluate`?

Who Writes `evaluate`?

- The implementor of `BooleanExpression`...if so, it should be inside the `BooleanExpression` class
- The client of `BooleanExpression`...if so, it should be outside the `BooleanExpression` class

The implementor of `BooleanExpression` should write the method, because then all the clients can use it.

That pesky `static`...

- If the implementor writes `evaluate`, then the method signature is:

  ```java
  public Boolean evaluate(Map<String, Boolean> assn)
  ```

- If the client writes `evaluate`, then the method signature is:

  ```java
  public static Boolean evaluate(    
      BooleanExpression e,           
      Map<String, Boolean> assn
  )
  ```
Write a method

```java
public static boolean canBeTrue(BooleanExpression b)
```
that returns true if it is possible for the input to evaluate to true and false otherwise.

Some examples:

- `a && b` → if we have `{a=true, b=true}`, then it is true.
- `a && !a` → no matter what `a` is, this will always be false.

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

- What are the choices we’re making incrementally?
- How do we “undo” a choice?
- What are the base case(s)?
Finally, Back To Recursive Backtracking. . .

canBeTrue

Write a method

    public static boolean canBeTrue(BooleanExpression b)

that returns true if it is possible for the input to evaluate to true and false otherwise.

Some examples:
- a && b → if we have {a=true, b=true}, then it is true.
- a && !a → no matter what a is, this will always be false.

To do recursive backtracking, we need to answer these questions:
- What are the choices we’re making incrementally?
  ... assignments of each variable to true/false
- How do we “undo” a choice?
  ... remove the assignment from the map
- What are the base case(s)?
  ... the assignment must be true/false
Uh Oh... How can we try assignments?

We don't have a way of passing assignments through to the function. How can we fix this?

**public/private pair!**

Public/Private Recursive Pair

```java
class PublicPrivatePair {
    public static boolean canBeTrue(BooleanExpression b)
    {
        // Implementation
    }

    private static boolean canBeTrue(
        BooleanExpression b,
        Map<String, Boolean> m
    )
    {
        // Implementation
    }
}
```
public static Map<String, Boolean> canBeTrue(BooleanExpression b) {
    Map<String, Boolean> assignmentMap = new TreeMap<>();
    if (canBeTrue(b, assignmentMap)) { // Some assignment works...
        return assignmentMap;
    }
    return null;
}

private static boolean canBeTrue(BooleanExpression b, Map<String, Boolean> m) {
    Boolean result = b.evaluate(m); // See if the current assignment works
    if (result != null) { // If we can’t answer, backtrack
        return result;
    }

    Set<String> variables = b.getVariables();
    variables.removeAll(m.keySet());
    for (String variable : variables) { // Try to assign any
        boolean[] choices = {true, false}; // variable we haven’t
        for (boolean assignment : choices) { // already assigned.
            m.put(variable, assignment);
            Boolean attempt = canBeTrue(b, m);
            if (attempt) { // If we found an
                return true; // assignment, we’re good.
            }
            m.remove(variable); // Otherwise, backtrack
        }
    }
    return false;
}
Solving `canBeTrue` quickly is the **most important** open problem in Computer Science.

If you solve this problem in $O(n^k)$ time for **any** $k$, the following happen:

- You get **one million** dollars.
- You get a PhD.
- You become the most famous Computer Scientist, pretty much ever
- You break all banks, credit cards, website encryption, etc.