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)

Client vs. Implementor, again

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.
Evaluating BooleanExpression

Remember when we took $(1+2)^3$ and evaluated it to 9 recursively? We can do a similar thing for BooleanExpressions:

Consider the BooleanExpression from above:

$"(!a \& \& b)"$

Suppose we know the following:

- $a$ is true.
- $b$ is false.

What does this expression evaluate to?

$(!a \& \& b) \rightarrow (!true \& \& false) \rightarrow (false \& \& false) \rightarrow false$

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.

Who Should Implement evaluate?

Who Writes evaluate?

- The implementor of BooleanExpression
- The client of BooleanExpression

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
  )
  ```

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?

```java
public/private pair!
```

Public/Private Recursive Pair

```java
public static boolean canBeTrue(BooleanExpression b) {
  Map<String, Boolean> assignments = new TreeMap<>();
  return canBeTrue(b, assignments);
}
```

```java
private static boolean canBeTrue(
  BooleanExpression b,
  Map<String, Boolean> assignmentMap
)
```

Finally, Back To Recursive Backtracking...

canBeTrue

Write a method

```java
public static void canBeTrue(BooleanExpression b)
```

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

Some examples:

- $a \& \& b \rightarrow$ if we have $(a=true, b=true)$, then it is true.
- $a \& \& !a \rightarrow$ 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

CanBeTrue Solution

```java
1  public static void canBeTrue(BooleanExpression b) {
2    Map<String, Boolean> assignmentMap = new HashMap<>();
3    if (canBeTrue(b, assignmentMap)) {
4      System.out.println("true");
5    }
6  }
7
8  public static void canBeTrue(BooleanExpression b, Map<String, Boolean> assignmentMap) {
9    // Try to assign any
10    for (String variable : assignments.keySet()) {
11        if (assignmentMap.containsKey(variable)) {
12          boolean[] choices = (true, false); // already assigned.
13          for (boolean assignment : choices) {
14            if (assignmentMap.get(variable) == assignment) {
15              assignmentMap.put(variable, assignment);
16              canBeTrue(b, assignmentMap);
17            } else {
18                assignmentMap.remove(variable); // Otherwise, backtrack
19            }
20          }
21        }
22    }
23  }
```
BTW, Why does this problem matter?

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.