Solution: Generate Parentheses

Let's solve the Generate Parentheses problem using the Subsets pattern.

We'll cover the following

Statement

For a given number, `n`, generate all combinations of balanced parentheses.

Constraints:

- $1 \leq$ `n` ≤ 13

Solution

Problems where we require constructing multiple valid answers, such as permutation of any n number or generating a long list of n pairs of balanced parentheses, are good examples to solve using the subsets pattern.

To solve this problem using the subsets pattern, we use a recursive algorithm. The purpose of this algorithm is to correctly generate all of the possible subsets of the parentheses combination.

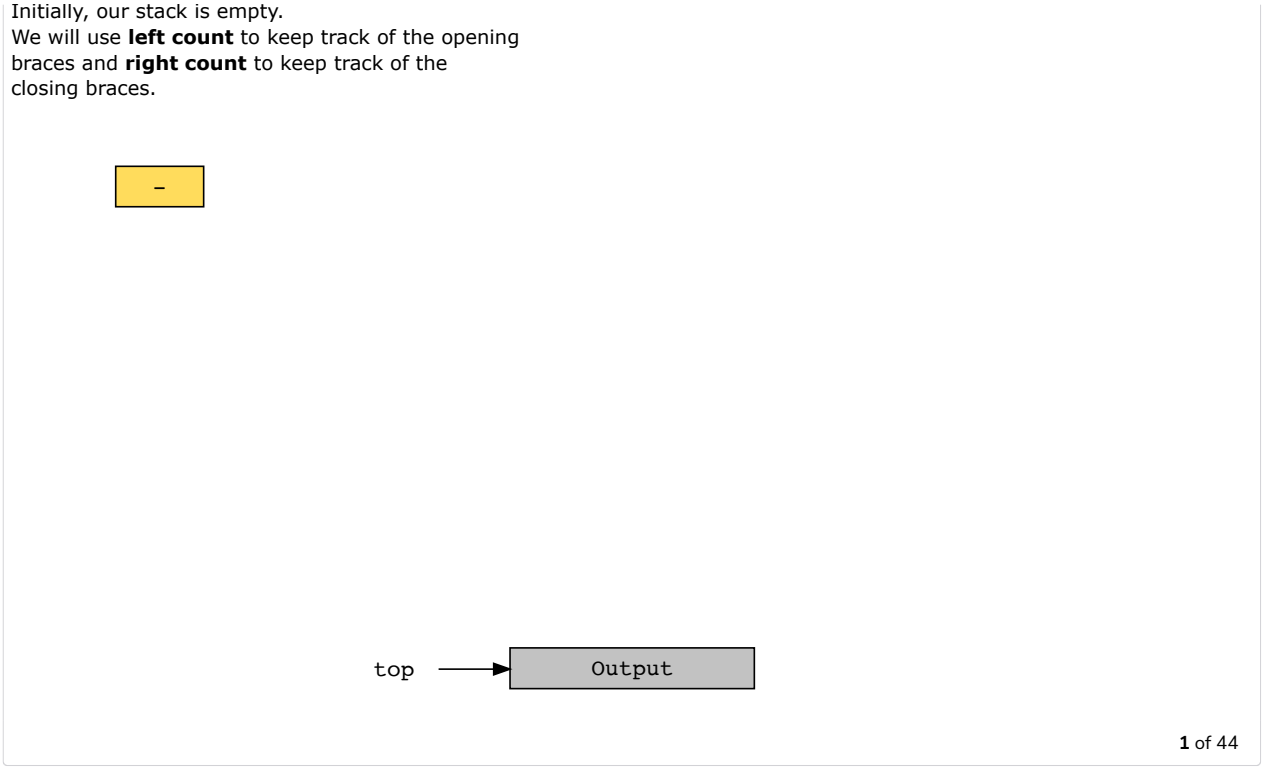
The key observation in constructing the solution is to keep track of the number of opening, `(`, and closing, `)`, parentheses. We'll use this count to add parentheses in a way that keeps the sequence balanced and valid.

- We only add an opening parenthesis if there is still one to add.
- We only add a closing parenthesis if the number of closing parentheses does not exceed the number of opening parentheses.

The basic algorithm is as follows:

1. Create a list to store all possible combinations of parentheses.
2. Call the `back_track` function for `n` pairs of parentheses, count of left parentheses, count of right parentheses, an empty list for output, and an empty list for the result.
3. If the count of opening and closing parentheses equals `n`, then we have a valid combination of parentheses, so we will append the string to our list.
4. Otherwise, we will check if the number of opening parentheses is less than `n`. If yes, we will add opening parentheses to our string and increment the count.
5. Lastly, we will check the count of closing parentheses. If the count is less than the count of opening parentheses, then we will add closing parentheses to our string and increment its count.

Let's see how this algorithm works for an example, where $n = 3$:



Let's look at the code for this solution below:

Java

```java
class AllBraces {
    // The recursive backtracking function
    static void backtrack(int n, int leftCount, int rightCount, ArrayList<Character> output, ArrayList<St

        // Base case where count of left and right braces is n
        if (leftCount >= n && rightCount >= n) {
            String outputStr = output.toString();
            result.add(outputStr.substring(1, outputStr.length()-1).replace(", ", ""));
        }

        // Case where we can still add left braces
        if (leftCount < n) {
            output.add('(');
            backtrack(n, leftCount + 1, rightCount, output, result);
            output.remove(output.size() - 1);
        }

        // Case where we add right braces if the current count
        // of right braces is less than the count of left braces
        if (rightCount < leftCount) {
            output.add(')');
            backtrack(n, leftCount, rightCount + 1, output, result);
            output.remove(output.size() - 1);
        }
    }

    static ArrayList<String> generateCombinations(int n) {
        ArrayList<String> result = new ArrayList<String>();
```

Generate Parentheses

Time complexity

As visualized in the slides above, it's easier to understand the problem by creating a tree. Given a tree with branching of b and a maximum depth of m, the maximum number of nodes in this tree would be the

following:

$$1 + b + b^2 + ... + b^{(m-1)}$$

This is the sum of a geometric sequence, which evaluates to the following:

$$\frac{(1 - b^m)}{(1 - b)}$$

For our problem here, the following is true:

- There is a branching of 2 because we only add an opening or a closing brace at each step.
- There is a maximum depth of $2n$ because the length of the output is the sum of n opening and n closing parentheses.

This leads us to a time complexity of $O(2^{(2n)})$, that is, $O(4^n)$.

> **Note:** This is not the tightest bound on time complexity because our algorithm rejects invalid combinations, significantly saving time. Without going into the proof, which is beyond the scope of this course, we'll note that the upper bound on the time complexity of this solution is $O(\frac{4^n}{\sqrt{n}})$.

Space complexity

The space complexity of this solution is $O(n)$, since the maximum size of the stack is $2n$.

Back
Generate Parentheses

Next
Find K-Sum Subsets

Mark as Completed