

Balanced Parentheses (hard)

We'll cover the following ^

- Problem Statement
- Try it yourself
- Solution
- Code
 - Time complexity
 - Space complexity
- Recursive Solution

Problem Statement

For a given number 'N', write a function to generate all combination of 'N' pairs of balanced parentheses.

Example 1:

```
Input: N=2
Output: (()), ()()
```

Example 2:

```
Input: N=3
Output: ((())), (()()), ((())(), ()(()), ()()())
```

Try it yourself

Try solving this question here:

Java Python3 JS C++

```
1 def generate_valid_parentheses(num):
2     result = []
3     # TODO: Write your code here
4     return result
5
6
7 def main():
8     print("All combinations of balanced parentheses are: " +
9         str(generate_valid_parentheses(2)))
10    print("All combinations of balanced parentheses are: " +
11        str(generate_valid_parentheses(3)))
12
13
14 main()
15
```

Run Save Reset

Solution

This problem follows the [Subsets](#) pattern and can be mapped to [Permutations](#). We can follow a similar BFS approach.

Let's take Example-2 mentioned above to generate all the combinations of balanced parentheses. Following a BFS approach, we will keep adding open parentheses '(' or close parentheses ')'. At each step we need to keep

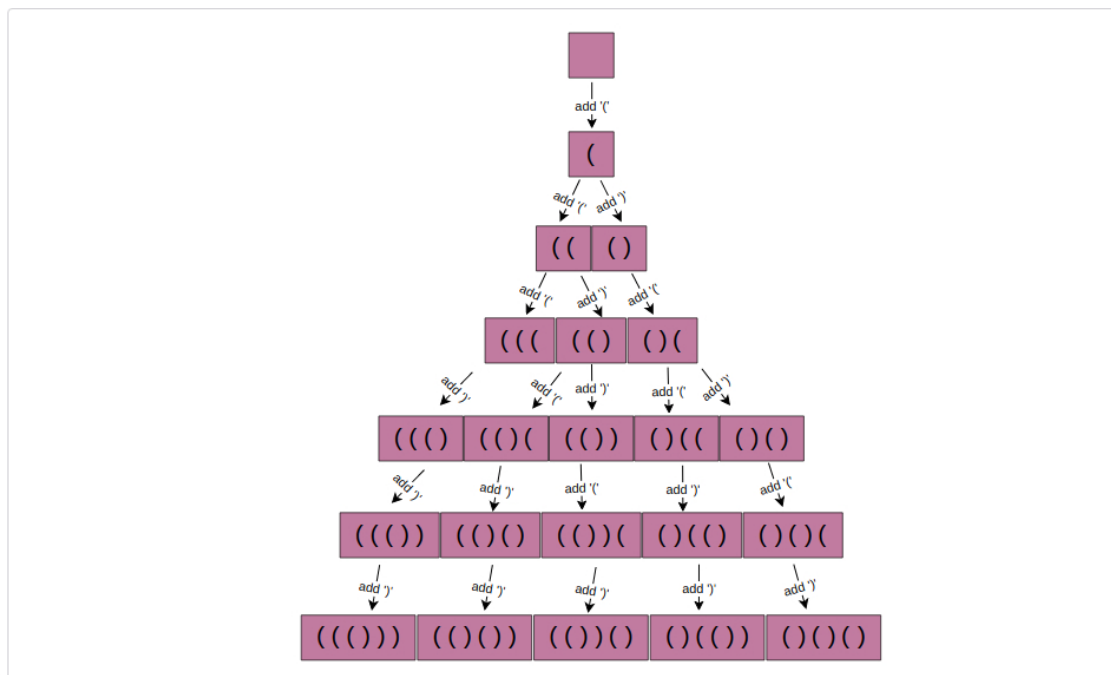
In this approach, we will keep adding open parentheses (() or close parentheses)). At each step we need to keep two things in mind:

- We can't add more than 'N' open parenthesis.
- To keep the parentheses balanced, we can add a close parenthesis `)` only when we have already added enough open parenthesis `(`. For this, we can keep a count of open and close parenthesis with every combination.

Following this guideline, let's generate parentheses for N=3:

1. Start with an empty combination: ""
2. At every step, let's take all combinations of the previous step and add (or) keeping the above-mentioned two rules in mind.
3. For the empty combination, we can add (since the count of open parenthesis will be less than 'N'. We can't add) as we don't have an equivalent open parenthesis, so our list of combinations will now be: "("
4. For the next iteration, let's take all combinations of the previous set. For "(" we can add another (to it since the count of open parenthesis will be less than 'N'. We can also add) as we do have an equivalent open parenthesis, so our list of combinations will be: "(", ")"
5. In the next iteration, for the first combination "(", we can add another (to it as the count of open parenthesis will be less than 'N', we can also add) as we do have an equivalent open parenthesis. This gives us two new combinations: "(((" and "(()". For the second combination "()", we can add another (to it since the count of open parenthesis will be less than 'N'. We can't add) as we don't have an equivalent open parenthesis, so our list of combinations will be: "((((", "(()(", "()"
6. Following the same approach, next we will get the following list of combinations: "((((", "(()(", "(()(", "()((", "((((", "(((())", "((()()", "()((((", "()()"
7. Next we will get: "(((())", "((()()", "(()()()", "()((()()", "((((()", "(((())()", "((()()()", "()(((())", "()()()"
8. Finally, we will have the following combinations of balanced parentheses: "(((())()", "((()()()()", "(()()()()()", "()((()()()()", "((((()()()", "(((())()()()", "((()()()()()", "()(((())()()()", "()()()()()"
9. We can't add more parentheses to any of the combinations, so we stop here.

Here is the visual representation of this algorithm:



Code

Here is what our algorithm will look like:

Java

Python3

C++

JS

```

1 from collections import deque
2
3
4 class ParenthesesString:
5     def __init__(self, str, openCount, closeCount):
6         self.str = str
7         self.openCount = openCount
8         self.closeCount = closeCount
9
10
11 def generate_valid_parentheses(num):
12     result = []
13     queue = deque()
14     queue.append(ParenthesesString("", 0, 0))
15     while queue:
16         ps = queue.popleft()
17         # if we've reached the maximum number of open and close parentheses, add to the result
18         if ps.openCount == num and ps.closeCount == num:
19             result.append(ps.str)
20         else:
21             if ps.openCount < num: # if we can add an open parentheses, add it
22                 queue.append(ParenthesesString(
23                     ps.str + "(", ps.openCount + 1, ps.closeCount))
24
25             if ps.openCount > ps.closeCount: # if we can add a close parentheses, add it
26                 queue.append(ParenthesesString(ps.str + ")",
27                     ps.openCount, ps.closeCount + 1))
28
29     return result
30
31
32 def main():
33     print("All combinations of balanced parentheses are: " +
34         str(generate_valid_parentheses(2)))
35     print("All combinations of balanced parentheses are: " +
36         str(generate_valid_parentheses(3)))
37
38
39 main()
40

```

Run

Save

Reset

Time complexity

Let's try to estimate how many combinations we can have for 'N' pairs of balanced parentheses. If we don't care for the ordering - *that) can only come after (* - then we have two options for every position, i.e., either put open parentheses or close parentheses. This means we can have a maximum of 2^N combinations. Because of the ordering, the actual number will be less than 2^N .

If you see the visual representation of Example-2 closely you will realize that, in the worst case, it is equivalent to a binary tree, where each node will have two children. This means that we will have 2^N leaf nodes and $2^N - 1$ intermediate nodes. So the total number of elements pushed to the queue will be $2^N + 2^N - 1$, which is asymptotically equivalent to $O(2^N)$. While processing each element, we do need to concatenate the current string with (or). This operation will take $O(N)$, so the overall time complexity of our algorithm will be $O(N * 2^N)$. This is not completely accurate but reasonable enough to be presented in the interview.

The actual time complexity ($O(4^n / \sqrt{n})$) is bounded by the [Catalan number](#) and is beyond the scope of a coding interview. See more details [here](#).

Space complexity

All the additional space used by our algorithm is for the output list. Since we can't have more than $O(2^N)$ combinations, the space complexity of our algorithm is $O(N * 2^N)$.

Recursive Solution

Here is the recursive algorithm following a similar approach:

JavaPythonC++JS

```
1 def generate_valid_parentheses(num):
2     result = []
3     parenthesesString = [''] * (2 * num)
4     generate_valid_parentheses_rec(num, 0, 0, parenthesesString, 0, result)
5     return result
6
7
8 def generate_valid_parentheses_rec(num, openCount, closeCount, parenthesesString, index, result):
9
10    # if we've reached the maximum number of open and close parentheses, add to the result
11    if openCount == num and closeCount == num:
12        result.append(''.join(parenthesesString))
13    else:
14        if openCount < num: # if we can add an open parentheses, add it
15            parenthesesString[index] = '('
16            generate_valid_parentheses_rec(
17                num, openCount + 1, closeCount, parenthesesString, index + 1, result)
18
19        if openCount > closeCount: # if we can add a close parentheses, add it
20            parenthesesString[index] = ')'
21            generate_valid_parentheses_rec(
22                num, openCount, closeCount + 1, parenthesesString, index + 1, result)
23
24
25 def main():
26     print("All combinations of balanced parentheses are: " +
27           str(generate_valid_parentheses(2)))
28     print("All combinations of balanced parentheses are: " +
29           str(generate_valid_parentheses(3)))
30
31
32 main()
33
```

RunSaveReset

[← Back](#)[Next →](#)

String Permutations by changing case...

Unique Generalized Abbreviations (ha...

✓ Completed

🚩 Report an Issue 🗨️ Ask a Question