25/03/2023

Generate Parantheses

We are given n pair of parantheses, write a function to generate all combinations of well formed parantheses.

n = 1 → ( ) } only 1 option

n = 2 → (( )), ( )( ) } only 2 options

n = 3 → ((( ))), (( )( )), ( )( )( ), ( )(( )), (( ))( )

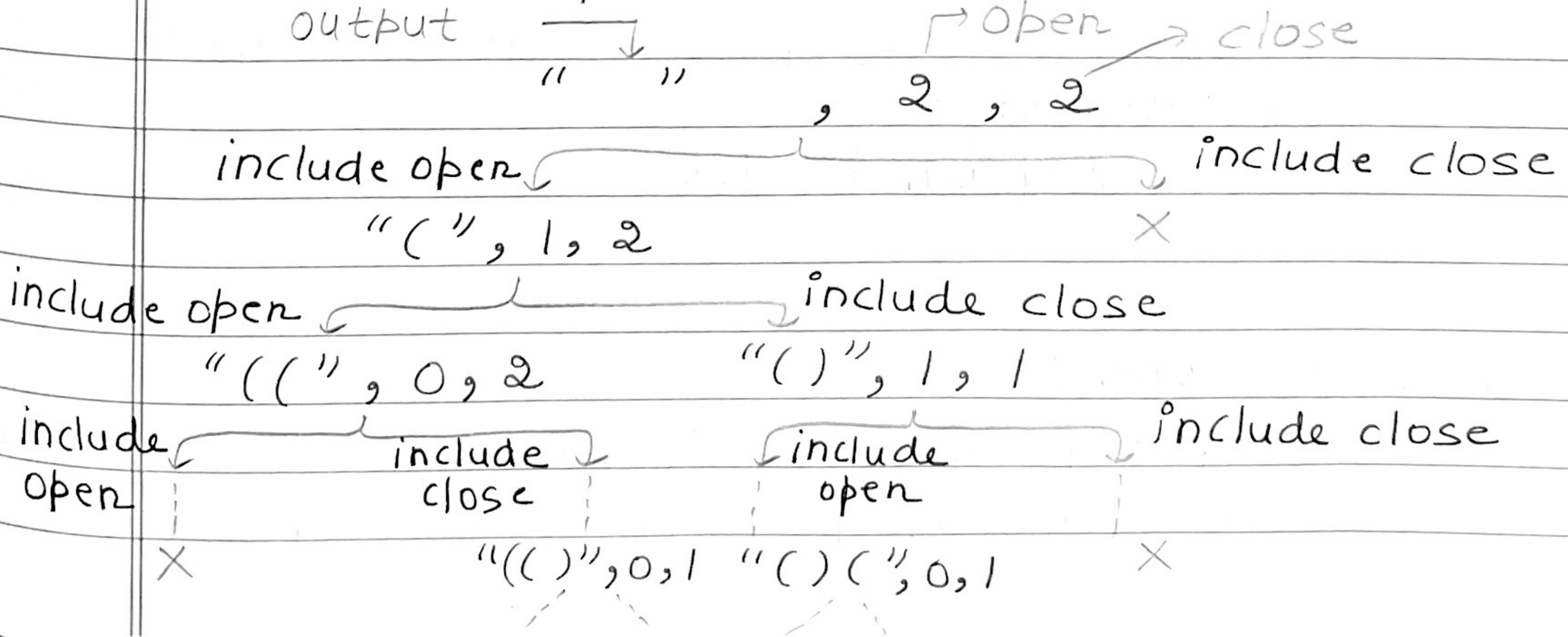
5 options are there

We can observe that if we are having n, then total brackets are 2 × n.

n = 3

$$\left.\begin{array}{l} (\rightarrow 3 \\ ) \rightarrow 3 \end{array}\right\} 3+3=6 \; \} \; 2\times 3 = 6$$

The above question is based on the include and exclude pattern.



We can only including the closing bracket, if on left count of opening bracket > count of closing bracket. But this has an issue that we will have to count on the left & hence we need to think of some condition in terms of remaining brackets. In left part formulae will be open >close & then only we will be including the closed bracket but in right part close > open & hence we will be using the closing bracket. When close == open, we don't have to add closing bracket.

Code

```
void solve (vector <string >& ans, int n, int open,
            int close, string output) {
 // Base case → Both brackets have finished
 if (open == 0 && close == 0) {
        ans.push_back (output);
        return;
 }
 // Include opening bracket if they exist.
 if (open > 0) {                        → Reduce count of opening bracket
      output.push_back ('('); // Push opening bracket
      solve (ans, n, open-1, close, output);
      // Backtracking → create original state
      output.pop_back();
 }
// Can we put closing bracket?
if (close > open) {
     output.push_back(')'); //Push closing bracket
     solve (ans, n, open, close-1, output);
                          ↳ Reduced count of closing
                            brackets
        //Backtracking
        output.pop_back();
    }
}
```

// Printing ans vector in main function

In main() print the ans vector of strings.

Note→ Understanding the close > open condition

1) This close & open is of the remaining brackets.
   This is very important to understand

On Left side

✓ open < close
    (()))→ invalid

✓ open == close
    (())→ no need to add bracket

✓ open > close
    (()   → closing bracket can be added now

(()

left side ↳ right side

remaining closing bracket = 1 } close > open
remaining opening bracket = 0

Hence we need to add the closing bracket & that's why close > open condition needs to be mentioned.

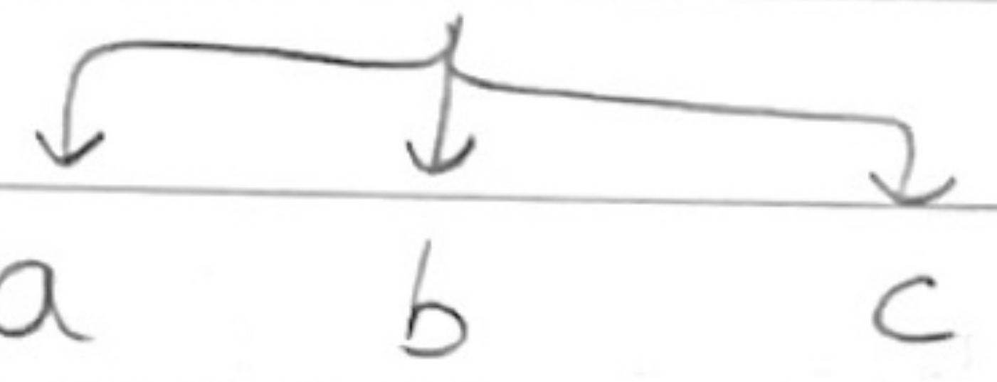
↱ Asked a lot in D.E. Show & Arcesium

Letter combination of phone number

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that number could represent. Return answer in any order.

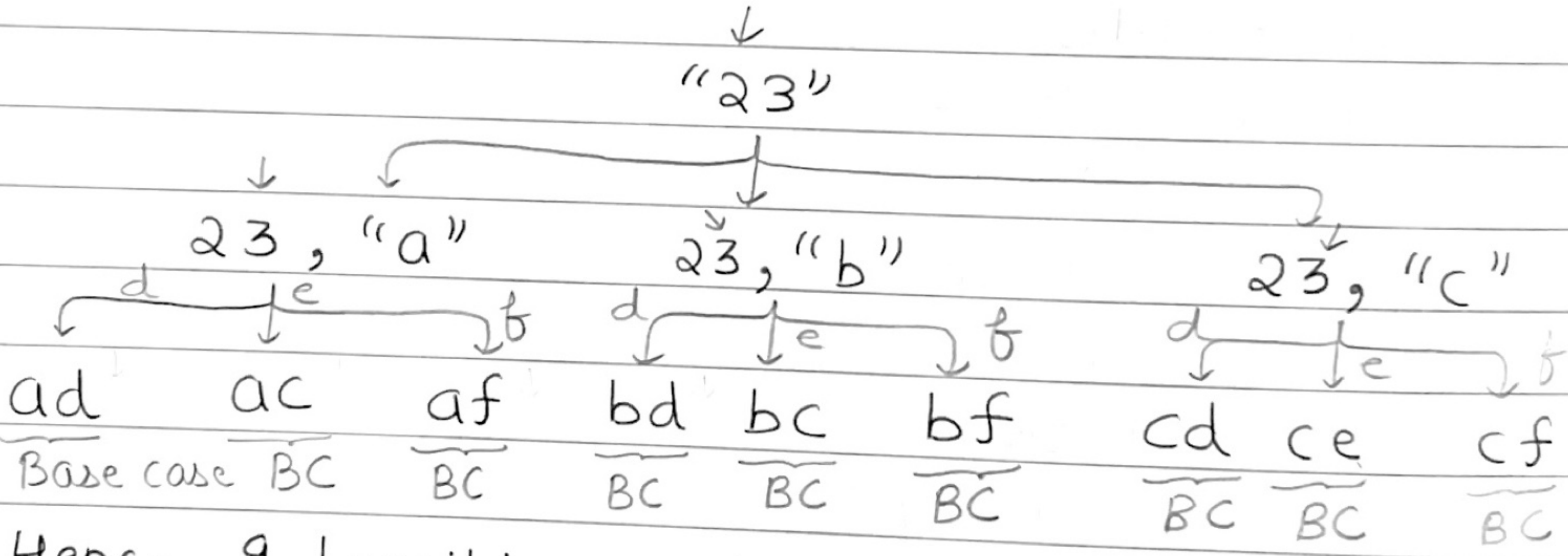
2 → abc　　6 → mno
3 → def　　7 → pqrs
4 → ghi　　8 → tuv　　} mapping
5 → jkl　　9 → wxyz

i/p → "2"



} 3 possible combinations

i/p → "23"



Hence 9 possible combinations are possible.

Code

```
void solve (vector <string> & ans , int index,
            string, output, string digits,
            vector <string> & mapping ) {
  → Can be handled in main function
  if (digits. length() = = 0) //Empty digits string
          return;
  if (index > = digits. length()) {
        ans. push_back (output);      } Base case
        return;
  }
```

```
//Solve 1 case & then recursion handles it
        int digit = digits [index] - '0' ;
                    character          used to convert
                                       to integer

        string value = mapping [digit] ; //Store the
                                           mapping

        // Traverse the mapping
        for (int i=0 ; i<value.length () ; i++){
            char ch = value [i]; // Store char of
                                    mapping in ch.

            output.push_back (ch) ; // Push that in
                                       O/p string.

            solve (ans, index+1, output, digits, mapping)
                        ↳ move forward in mapping

            // Backtracking
            output.pop_back () ; // If we have used
    }       push back, then use pop-back also.
}
```