Solution: Letter Combinations of a Phone Number

Let's solve the Letter Combinations of a Phone Number problem using the Subsets pattern.

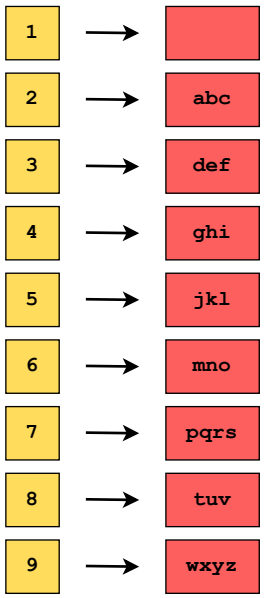
We'll cover the following

Statement

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

A mapping of digits to letters is given below.

> **Note**: 1 doesn't map to any letter.



Constraints:

- $0 \leq$ `digits.length` ≤ 4
- `digits[i]` is a digit in the range $[2, 9]$

Pattern: Subsets

This problem lends itself naturally to the subsets pattern. To solve this problem, we can use a backtracking algorithm as the solution template to correctly generate all the possible combinations.

Solution

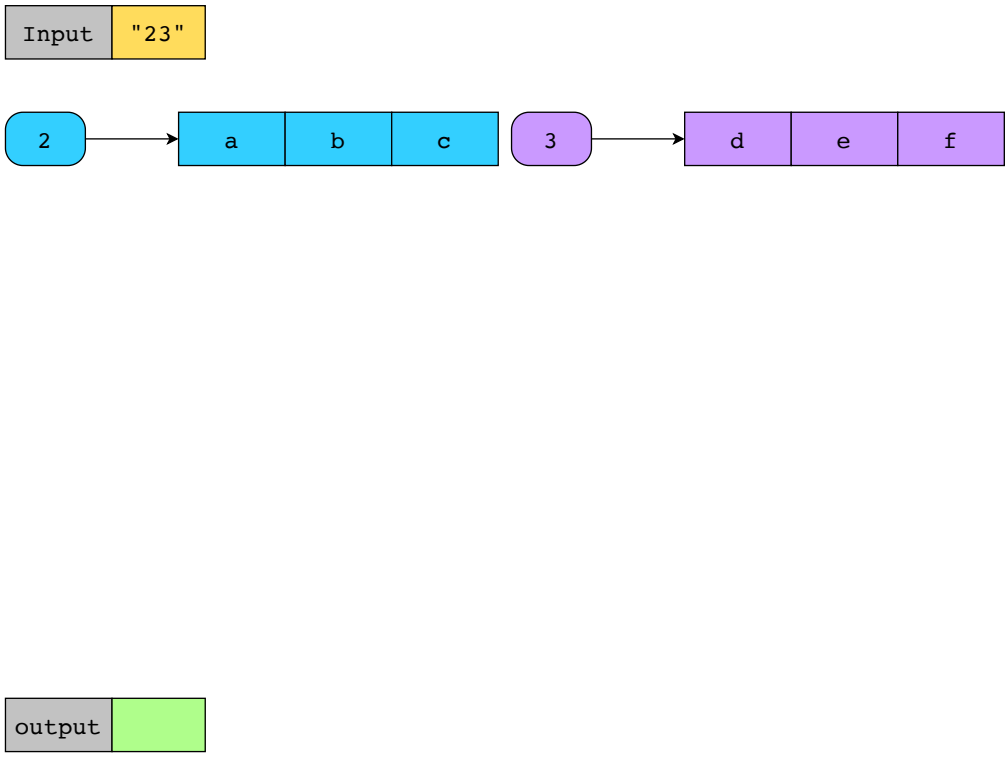
To generate all possible combinations of letters corresponding to each digit in the string, we can recursively move to the next index, retrieving the respective letters for each digit. Once the length of the current generated combination is equal to the length of the digit string, we will append the generated combination to the combinations list. After generating a combination, we then remove the last letter from the current generated combination list and explore other combinations by iterating through the remaining letters. Finally, we will return all possible combinations of letters from the given digit string.

For example, let's consider the input digits "23". We start with an empty combination list and the initial index of 0. For the first digit "2", the corresponding letters are ["a", "b", "c"]. We add "a" to the combination and recursively call the function with the next index of 1. For the second digit "3", the corresponding letters are ["d", "e", "f"]. We add "d" to the combination and recursively call the function with the next index of 2. At this point, the length of the combination is equal to the length of the input string, so we join the combination into a string, "ad", and append it to the list of combinations. Then, we remove the last letter "d" from the combination and move on to the next letter, "e", in the letters corresponding to digit "3". We repeat this process until we have explored all possible combinations.

To generate all the possible letter combinations, we will create a function, `letterCombinations`, which takes a string, `digits`, as input and returns a list of all possible letter combinations. The function will perform the below-mentioned steps:

1. The function first checks if the input string is empty. If it is empty, we will return an empty list immediately.
2. Create a hash map `digitsMapping`, which maps each digit to a list of corresponding letters. For example, "2" corresponds to letters "a", "b", and "c".
3. The `backtrack` function is then called to generate the combinations recursively. It takes the parameters, `index` (the current index in the digits string), `path` (the current combination of letters being built), `digits` (the input string of digits), `letters` (the mapping of digits to letters) and `combinations` (a list to store the generated combinations). Within the `backtrack` function, we will perform the following steps:
    - If the length of `path` is equal to the length of `digits`, it means we have a complete combination. The `path` is joined into a string and appended to the `combinations` list.
    - Otherwise, the function retrieves the list of possible letters corresponding to the digit at the current index. It then iterates through each letter and performs the following steps:
        - It adds the letter to the `path`.
        - It recursively calls `backtrack` with the updated index and path to move on to the next digit.
        - After the recursive call, the letter is removed from the `path` to backtrack and explore other letter combination possibilities.
4. Finally, the `combinations` list is returned as a result that contains all possible letter combinations made from the numbers in the string.

The slides below illustrate how we'd like the algorithm to run:



We will generate all possible letter combinations of length 2, corresponding to the input string, “23”. We start with exploring the letters corresponding to the first digit, i.e., 2.

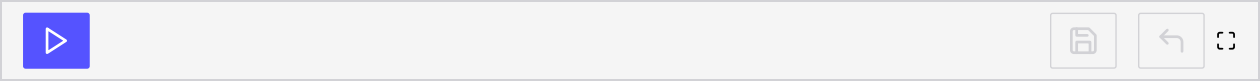
Let’s look at the code for this solution below:

Java

```java
import java.util.*;

class LetterCombinations {
    // Use backtrack function to generate all possible combinations
    public void backtrack(int index, StringBuilder path, String digits, Map<Character, String[]> letters,
        // If the length of path and digits is same,
        // we have a complete combination
        if (path.length() == digits.length()) {
            // Join the path list into a string and add it to combinations list
            combinations.add(path.toString());
            return; // Backtrack;
        }
        // Get the list of letters using the index and digits[index]
        char digit = digits.charAt(index);
        String[] possibleLetters = letters.get(digit);
        for (String letter : possibleLetters) {
            // Add the current letter to the path
            path.append(letter);
            // Recursively explore the next digit
            backtrack(index + 1, path, digits, letters, combinations);
            // Remove the current letter from the path before backtracking to explore other combinations
            path.deleteCharAt(path.length() - 1);
        }
    }

        List<String> combinations = new ArrayList<>();
        // If the input is empty, immediately return an empty answer array
```



Letter Combinations of a Phone Number

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. The backtracking approach used in the solution considers a digit as the starting point and generates all possible combinations with that letter.
2. The base case for the backtracking function will be that if our current combination of letters has the same length as the input digits, the iteration is complete.
3. Otherwise, we find all the possible combinations of letters that correspond with the current digit.

Time complexity

We designate the number of input digits as n and use k to represent the maximum number of letters associated with any given digit in our mapping. Generating the letter combinations takes $O(k^n \cdot n)$ time since there are k^n possible combinations for every digit.

Space complexity

The algorithm uses $O(n \cdot k)$ space, where n is the total number of input digits, and k is the maximum number of letters mapped to any digit.

Our recursive solution takes up space on the call stack, with n being the maximum depth of the stack, corresponding to the number of input digits. At each level in the stack, a list of up to k letters is maintained. Therefore, the space complexity is $O(n \cdot k)$.