**Letter Combinations of a Phone Number**

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

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

A picture containing electronics, telephone, calculator, orange

Description automatically generated

**Example 1:**

**Input:** digits = "23"

**Output:** ["ad","ae","af","bd","be","bf","cd","ce","cf"]

**Example 2:**

**Input:** digits = ""

**Output:** []

**Example 3:**

**Input:** digits = "2"

**Output:** ["a","b","c"]

**Constraints:**

* 0 <= digits.length <= 4
* digits[i] is a digit in the range ['2', '9'].
* Solution
* Overview
* One of the first things you should always do is look at the constraints. Quite often, you can figure out what sort of approach needs to be taken simply from looking at the input size. In an interview, asking your interviewer about the constraints will also show your attention to detail - on top of giving you information.
* In this particular problem, the length of the input is **extremely** small, 0 <= digits.length <= 4. With such small input sizes, we can safely assume that a brute force solution in which we generate all combinations of letters will be accepted.
* Whenever you have a problem where you need to generate all combinations/permutations of some group of letters/numbers, the first thought you should have is backtracking. If you're new to backtracking, check out our [backtracking explore card](https://leetcode.com/explore/featured/card/recursion-ii/472/backtracking/). Backtracking algorithms can often keep the *space complexity* linear with the input size.
* Approach 1: Backtracking
* **Intuition**
* There aren't any smart tricks needed for this problem - the hard part is just figuring out how to correctly generate all possible combinations, and to do this using a standard backtracking algorithm template. Let's break down the problem, by starting with an input that is only 1-digit long, for example digits = "2". This example is trivial - just generate all letters that correspond with digit = "2", which would be ["a", "b", "c"].
* What if instead we had a 2-digit long input, digits = "23"? Imagine taking each letter of digit = "2" as a *starting point*. That is, **lock the first letter in**, and solve all the possible combinations that **start with that letter**. If our first letter will always be "a", then the problem is trivial again - it's the 1-digit case, and all we have to do is generate all the letters corresponding with digit = "3", and add that to "a", to get ["ad", "ae","af"]. This was easy because we ignored the first letter, and said it will always be "a". But we know how to generate all the first letters too - it's the 1-digit case which we already solved to be ["a", "b", "c"].
* As you can see, solving the 1-digit case is trivial, and solving the 2-digit case is just solving the 1-digit case twice. The same reasoning can be extended to n digits. For the 3-digit case, solve the 2-digit case to generate all combinations of the first 2 letters, and then solve the 1-digit case for the final digit. Now that we know how to solve the 3-digit case, to solve the 4-digit case, solve the 3-digit case for all combinations of the first 3 letters, and then solve the 1-digit case for the final digit. We could extend this to infinity, but, don't worry, for this problem we're finished after 4.

**Algorithm**

As mentioned previously, we need to **lock-in** letters when we generate new letters. The easiest way to save state like this is to use recursion. Our algorithm will be as follows:

1. If the input is empty, return an empty array.
2. Initialize a data structure (e.g. a hash map) that maps digits to their letters, for example, mapping "6" to "m", "n", and "o".
3. Use a backtracking function to generate all possible combinations.
   * The function should take 2 primary inputs: the current combination of letters we have, path, and the index we are currently checking.
   * As a base case, if our current combination of letters is the same length as the input digits, that means we have a complete combination. Therefore, add it to our answer, and backtrack.
   * Otherwise, get all the letters that correspond with the current digit we are looking at, digits[index].
   * Loop through these letters. For each letter, add the letter to our current path, and call backtrack again, but move on to the next digit by incrementing index by 1.
   * Make sure to remove the letter from path once finished with it.

**Implementation**

class Solution {

private List<String> combinations = new ArrayList<>();

private Map<Character, String> letters = Map.of(

'2', "abc", '3', "def", '4', "ghi", '5', "jkl",

'6', "mno", '7', "pqrs", '8', "tuv", '9', "wxyz");

private String phoneDigits;

public List<String> letterCombinations(String digits) {

// If the input is empty, immediately return an empty answer array

if (digits.length() == 0) {

return combinations;

}

// Initiate backtracking with an empty path and starting index of 0

phoneDigits = digits;

backtrack(0, new StringBuilder());

return combinations;

}

private void backtrack(int index, StringBuilder path) {

// If the path is the same length as digits, we have a complete combination

if (path.length() == phoneDigits.length()) {

combinations.add(path.toString());

return; // Backtrack

}

// Get the letters that the current digit maps to, and loop through them

String possibleLetters = letters.get(phoneDigits.charAt(index));

for (char letter: possibleLetters.toCharArray()) {

// Add the letter to our current path

path.append(letter);

// Move on to the next digit

backtrack(index + 1, path);

// Backtrack by removing the letter before moving onto the next

path.deleteCharAt(path.length() - 1);

}

}

}

**Complexity Analysis**

* Time complexity: O(4^N \cdot N)*O*(4*N*⋅*N*), where N*N* is the length of digits. Note that 44 in this expression is referring to the maximum value length in the hash map, and ***not*** to the length of the input.

The worst-case is where the input consists of only 7s and 9s. In that case, we have to explore 4 additional paths for every extra digit. Then, for each combination, it costs up to N*N* to build the combination. This problem can be generalized to a scenario where numbers correspond with up to M*M* digits, in which case the time complexity would be O(M^N \cdot N)*O*(*MN*⋅*N*). For the problem constraints, we're given, M = 4*M*=4, because of digits 7 and 9 having 4 letters each.

* Space complexity: O(N)*O*(*N*), where N*N* is the length of digits.

Not counting space used for the output, the extra space we use relative to input size is the space occupied by the recursion call stack. It will only go as deep as the number of digits in the input since whenever we reach that depth, we backtrack.

As the hash map does not grow as the inputs grows, it occupies O(1)*O*(1) space.