dev.to

Solution: Letter Combinations of a Phone Number

5-7 minutes

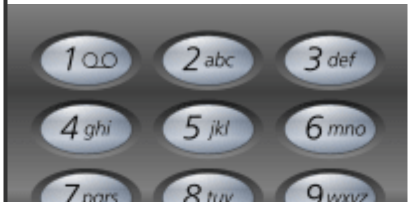
This is part of a series of Leetcode solution explanations (index). If you liked this solution or found it useful, ***please like*** *this post and/or* ***upvote*** *my solution post on Leetcode's forums.*

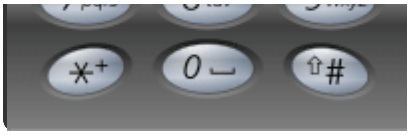
[Leetcode Problem #17 (*Medium*): Letter Combinations of a Phone Number](Leetcode Problem #17)

Description:

(*Jump to*: *Solution Idea* || *Code*: *JavaScript* | *Python* | *Java* | *C++*)

> 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 digit to letters (just like on the telephone buttons) is given below. Note that `1` does not map to any letters.





Examples:

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']`.

Idea:

(*Jump to*: *Problem Description* || *Code*: *JavaScript* | *Python* | *Java* | *C++*)

Since each digit can possibly mean one of several characters, we'll need to create code that branches down the different paths as we iterate through the input digit string (**D**).

This quite obviously calls for a **depth-first search** (**DFS**) approach as we will check each permutation of characters and store them in our answer array (**ans**). For a DFS approach we can use one of several options, but a **recursive** solution is generally the cleanest.

But first, we'll need to set up a lookup table (**L**) to convert a digit to its possible characters. Since the digits are actually low-indexed integers, we can actually choose between an **array** or **map/dictionary** here with little difference.

For our DFS function (**dfs**), we'll have to feed it the current position (**pos**) in **D** as well as the string (**str**) being built. The function will also need to have access to **D**, **L**, and **ans**.

The DFS function itself is fairly simple. It will push a completed **str** onto **ans**, otherwise it will look up the characters that match the current **pos**, and then fire off new recursive functions down each of those paths.

Once we're done, we should be ready to **return ans**.

Implementation:

Javascript and Python will have scoped access to **D**, **L**, and **ans** inside **dfs**, so won't need to pass in references via arguments.

Java should make sure to use a **char[][]** and a **StringBuilder** for better performance here.

Javascript Code:

(*Jump to*: [*Problem Description*]() || [*Solution Idea*]())

```
const L = {'2':"abc",'3':"def",'4':"ghi",'5':"jkl",
     '6':"mno",'7':"pqrs",'8':"tuv",'9':"wxyz"}

var letterCombinations = function(D) {
    let len = D.length, ans = []
    if (!len) return []
    const bfs = (pos, str) => {
        if (pos === len) ans.push(str)
        else {
            let letters = L[D[pos]]
            for (let i = 0; i < letters.length; i++)
                bfs(pos+1,str+letters[i])
        }
    }
    bfs(0,"")
    return ans
};
```

Python Code:

(*Jump to*: [*Problem Description*]() || [*Solution Idea*]())

```
L = {'2':"abc",'3':"def",'4':"ghi",'5':"jkl",
     '6':"mno",'7':"pqrs",'8':"tuv",'9':"wxyz"}
```

```
class Solution:
    def letterCombinations(self, D: str) ->
List[str]:
        lenD, ans = len(D), []
        if D == "": return []
        def bfs(pos: int, st: str):
            if pos == lenD: ans.append(st)
            else:
                letters = L[D[pos]]
                for letter in letters:
                    bfs(pos+1,st+letter)
        bfs(0,"")
        return ans
```

Java Code:

(*Jump to*: *Problem Description* || *Solution Idea*)

```
class Solution {
    final char[][] L = {{},{},{'a','b','c'},
{'d','e','f'},{'g','h','i'},{'j','k','l'},
    {'m','n','o'},{'p','q','r','s'},{'t','u','v'},
{'w','x','y','z'}};

    public List<String> letterCombinations(String D)
{
        int len = D.length();
        List<String> ans = new ArrayList<>();
        if (len == 0) return ans;
        bfs(0, len, new StringBuilder(), ans, D);
        return ans;
    }

    public void bfs(int pos, int len, StringBuilder sb, List<String> ans, String D) {
        if (pos == len) ans.add(sb.toString());
        else {
            char[] letters = L[Character.getNumericValue(D.charAt(pos))];
            for (int i = 0; i < letters.length; i++)
                bfs(pos+1, len, new StringBuilder(sb).append(letters[i]), ans, D);
        }
    }
}
```

C++ Code:

(*Jump to*: [*Problem Description*]() || [*Solution Idea*]())

```
unordered_map<char, string> L({{'2',"abc"},{'3',"def"},{'4',"ghi"},
    {'5',"jkl"},{'6',"mno"},{'7',"pqrs"},{'8',"tuv"},{'9',"wxyz"}});

```

```
class Solution {
public:
    vector<string> letterCombinations(string D) {
        int len = D.size();
        vector<string> ans;
        if (!len) return ans;
        bfs(0, len, "", ans, D);
        return ans;
    }

    void bfs(int pos, int &len, string str,
vector<string> &ans, string &D) {
        if (pos == len) ans.push_back(str);
        else {
            string letters = L[D[pos]];
            for (int i = 0; i < letters.size(); i++)
                bfs(pos+1, len, str+letters[i], ans,
D);
        }
    }
};
```