**QPREP1-LETTER COMBINATIONS OF PHONE NUMBER**

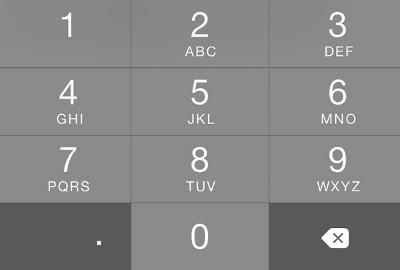
**Module Introduction**

Write a program to take a number sequence as input and output all possible letter combinations that can be represented by that number sequence on the phone.

**Objective**

Given an input string containing digits between 2 and 9 (both included), find and print all the letter combinations that number could represent.

The mapping can be seen here



Note: The output needs to be lexicographically sorted. Some digits can represent 4 characters.

**Examples**

**Example 1**

Input: 34

Output: dg dh di eg eh ei fg fh fi

**Example 2**

Input: 234

Output: adg adh adi aeg aeh aei afg afh afi bdg bdh bdi beg beh bei bfg bfh bfi cdg cdh cdi ceg ceh cei cfg cfh cfi

***SOLUTION STEPS FROM NEXT PAGE:***

**Write down at least 3 examples in the following format. Kindly stick to the format.**

**Suggestion:**

EXAMPLE#1

INPUT: 234

OUTPUT: adg adh adi aeg aeh aei afg afh afi bdg bdh bdi beg beh bei bfg bfh bfi cdg cdh cdi ceg ceh cei cfg cfh cfi

EXAMPLE#2

INPUT: 987 OUTPUT: wtp wtq wtr wts wup wuq wur wus wvp wvq wvr wvs xtp xtq xtr xts xup xuq xur xus xvp xvq xvr xvs ytp ytq ytr yts yup yuq yur yus yvp yvq yvr yvs ztp ztq ztr zts zup zuq zur zus zvp zvq zvr zvs

EXAMPLE#3

INPUT: 3

OUTPUT: d e f

**Detail your problem understanding here**

**Suggestion:**

Each digit from 2 to 9 represents certain letters on the phone’s keypad. 1 and 0 don’t represent any letters.

The input is a string with one or more digits between 2 to 9. E.g. 34, 265 etc.

Each digit in the input string can be replaced with one of the letters it stands for, thus generating a combination of letters to represent the input number string.

We need to generate all possible combinations of such letters.

**Does this problem follow a known algorithmic pattern or standard application of a data structure? If there are multiple approaches, which one would you choose and why? Write down your chosen approach in 2-3 sentences like you would explain to a 10 year old.**

The problem is a standard string permutation problem with an extra constraint that the letters have to be chosen based on keys pressed.

Recursion is a good way to address this problem.

**Write the pseudocode here in plain English**

Create static mapping of each digit to its corresponding letters

Create an empty list of strings that will be populated with the output

generateLetterCombinations(numberString) {

Handle any input validation.

Invoke function to generate combinations which will be a recursive function

recursivelyGenerateCombinations("", numberString)

return the updated list of output strings

}

recursivelyGenerateCombinations(stringToPrepend, inputNumberString) {

Terminating condition

Return If inputNumberString size is 0, add the stringToPrepend to the output list

getFirstDigit(inputNumberString)

getDigitToLettersMapping(firstDigit)

For each letter the firstDigit can represent

recursivelyGenerateCombinations(stringToPrepend+letter, inputNumberString-firstDigit)

}

**Can you specify a few boundary or edge cases here?**

INPUT: 12

OUTPUT:

INPUT: 1

OUTPUT:

INPUT: 0

OUTPUT:

INPUT: 22

OUTPUT: aa ab ac ba bb bc ca cb cc

**Write the functions you would create here**

String getDigitToLettersMapping(String digit)

List letterCombinations(String numberString)

recursivelyGenerateCombinations(String stringToPrepend, String inputNumberString)

Character getFirstDigit(String numberString)

#### Summary

Starting with a brief explanation of the problem statement, followed by pseudocode and then implementing the solution helps you approach the problem in a systematic way. This methodology helps with easy as well as hard problems.

**Time Complexity: O(3^N \* 4^M)**

Where N is the number of digits in the input that maps to 3 letters (e.g. 2, 3, 4, 5, 6, 8) and M is the number of digits in the input that maps to 4 letters (e.g. 7, 9), and N+M is the total number digits in the input. It will be O((N+M) \* (3^N \* 4^N)) if you consider the time taken to construct the strings.

**Space Complexity: O(3^N \* 4^M)**

Since we have to store 3^N \* 4^M solutions

#### Concepts

Concepts covered in this Module

* Strings
* Recursion

Similar problems

* <https://leetcode.com/problems/generate-parentheses/>
* <https://leetcode.com/problems/combination-sum/>
* <https://leetcode.com/problems/binary-watch/>

### *References*

* [*https://www.cs.cmu.edu/~adamchik/15-121/lectures/Recursions/recursions.html*](https://www.cs.cmu.edu/~adamchik/15-121/lectures/Recursions/recursions.html)
* [*https://hackernoon.com/20-string-coding-interview-questions-for-programmers-6b6735b6d31c*](https://hackernoon.com/20-string-coding-interview-questions-for-programmers-6b6735b6d31c)

#### Good habits

Think about these for your solution:

* Comments - have you used comments in a way that others can understand this code?
* Test Cases - Are most of the scenarios/corner cases/boundary conditions handled in the solution?
* Naming Convention - Are the variables and functions named sensibly and with uniform convention?
* Modular Functions - Has the solution been addressed using concise functions? Will these functions work without any changes if they are to be used in another problem?
* Optimization - Analyze the Time Complexity and Space Complexity for your solution. Has the solution been optimized or did it use the brute force method? Is further optimization desirable/possible?
* Data Structures - Has the optimal/appropriate data structure been used?

SOLUTION:

APPROACH 1:

**Approach:** The approach is slightly different from the approach in the other article. Suppose there are n keys which are pressed (a1 a2 a3 ..an). Find all the words that can be formed using (a2 a3 ..an). Suppose 3 characters can be generated by pressing a1 then for every character concatenate the character before all the words and insert them to the list.

**For Example:**

*If the keypress is 12  
The characters that can be formed by pressing 1 is a, b, c and by pressing 2 characters d, e, f can be formed.  
So all the words that can be formed using 2 are [d, e, f]  
So now concatenate ‘a’ with all words returned so, the list is [ad, ae, af] similarly concatenate b and c. So the list becomes [ad, ae, af, bd, be, bf, cd, ce, cf].*

**Algorithm:**

1. Write a recursive function that accepts key press string and returns all the words that can be formed in an [Array list](https://www.geeksforgeeks.org/arraylist-in-java/).
2. If the length of the given string is 0 then return Arraylist containing empty string.
3. Else recursively call the function with a string except the first character of original string, i.e string containing all the characters from index 1 to n-1. and store the arraylist returned, *list* and create a new arraylist *ans*
4. Get the character set of the first character of original string, *CSet*
5. For every word of the *list* run a loop through the *Cset* and concatenate the charcter of *Cset* infront of the word of *list* and insert them in the *ans* arraylist.
6. Return the arraylist, *ans*.

**Implementation:**

|  |
| --- |
| // Java implementation of the approach  import java.util.ArrayList;    public class GFG {        // String array to store keypad characters      static final String codes[]          = { " ", "abc", "def",              "ghi", "jkl", "mno",              "pqr", "stu", "vwx",              "yz" };        // Function that returns an Arraylist      // which contains all the generated words      public static ArrayList<String> printKeyWords(String str)      {            // If str is empty          if (str.length() == 0) {              ArrayList<String> baseRes = new ArrayList<>();              baseRes.add("");                // Return an Arraylist containing              // empty string              return baseRes;          }            // First character of str          char ch = str.charAt(0);            // Rest of the characters of str          String restStr = str.substring(1);            ArrayList<String> prevRes = printKeyWords(restStr);          ArrayList<String> Res = new ArrayList<>();            String code = codes[ch - '0'];            for (String val : prevRes) {                for (int i = 0; i < code.length(); i++) {                  Res.add(code.charAt(i) + val);              }          }          return Res;      }        // Driver code      public static void main(String[] args)      {          String str = "23";            // Print all the possible words          System.out.println(printKeyWords(str));      }  } |

**Complexity Analysis:**

* **Time Complexity:** O(3n).  
  Though the recursive function runs n times. But the size of the arraylist grows exponentially. So there will be around 3n elements in the arraylist. Therefore, traversing them will take 3n time.
* **Space Complexity:**O(3n).  
  Space required to store all words is O(3n). As there will be around 3n words in the output.

APPROACH 2:

**Example:**

If the number is 23,

Then for 2, the alphabets are a, b, c

So 3 recursive function will be called

with output string as a, b, c respectively

and for 3 there are 3 alphabets d, e, f

So, the output will be ad, ae and af for

the recursive function with output string.

Similarly, for b and c, the output will be:

bd, be, bf and cd, ce, cf respectively.

**Algorithm:**

1. Map the number with its string of probable alphabets, i.e 2 with “abc”, 3 with “def” etc.
2. create a recursive function which takes following parameters, output string, number array, current index and length of number array
3. If the current index is equal to the length of number array then print the output string.
4. Extract the string at *digit[current\_index]* from the Map, where digit is the input number array.
5. Run a loop to traverse the string from start to end
6. For every index again call the recursive function with the output string concatenated with the ith character of the string and the current\_index + 1.

**Implementation:** Note that input number is represented as an array to simplify the code.

|  |
| --- |
| #include <stdio.h>  #include <string.h>    // hashTable[i] stores all characters that correspond to digit i in phone  const char hashTable[10][5] = {"", "", "abc", "def", "ghi", "jkl",                                 "mno", "pqrs", "tuv", "wxyz"};    // A recursive function to print all possible words that can be obtained  // by input number[] of size n.  The output words are one by one stored  // in output[]  void  printWordsUtil(int number[], int curr\_digit, char output[], int n)  {      // Base case, if current output word is prepared      int i;      if (curr\_digit == n)      {          printf("%s ", output);          return ;      }        // Try all 3 possible characters for current digir in number[]      // and recur for remaining digits      for (i=0; i<strlen(hashTable[number[curr\_digit]]); i++)      {          output[curr\_digit] = hashTable[number[curr\_digit]][i];          printWordsUtil(number, curr\_digit+1, output, n);          if (number[curr\_digit] == 0 || number[curr\_digit] == 1)              return;      }  }    // A wrapper over printWordsUtil().  It creates an output array and  // calls printWordsUtil()  void printWords(int number[], int n)  {      char result[n+1];      result[n] ='\0';      printWordsUtil(number, 0, result, n);  }    //Driver program  int main(void)  {      int number[] = {2, 3, 4};      int n = sizeof(number)/sizeof(number[0]);      printWords(number, n);      return 0;  } |

**Output:**

adg adh adi aeg aeh aei afg afh afi bdg

bdh bdi beg beh bei bfg bfh bfi cdg cdh

cdi ceg ceh cei cfg cfh cfi

Process returned 0 (0x0) execution time : 0.025 s

Press any key to continue.

**Complexity Analysis:**

* **Time Complexity:**O(4n), where n is number of digits in input number.  
  Each digit of a number has 3 or 4 alphabets, so it can be said that each digit has 4 alphabets as options. If there are n digits then there are 4 options for first digit and for each alphabet of first digit there are 4 options in second digit, i.e for every recursion 4 more recursion is called (if it does not match the base case). So the time complexity is O(4n).
* **Space Compelxity:**O(1).  
  As no extra space is needed.

APPROACH 3:

**Example:**

***Input:****arr[] = {2, 3}****Output:****ad ae af bd be bf cd ce cf*

***Input:****arr[] = {9}****Output:****w x y z*

**Approach:** Now let us think how we would approach this problem without doing it in an iterative way. A recursive solution is intuitive and common. We keep adding each possible letter recursively and this will generate all the possible strings.

Let us think about how we can build an iterative solution using the recursive one. Recursion is possible through the use of a stack. So if we use a stack instead of a recursive function will that be an iterative solution? One could say so speaking technically but we then aren’t really doing anything different in terms of logic.

A Stack is a LIFO DS. Can we use another Data structure? What will be the difference if we use a FIFO DS? Let’s say a queue. Since BFS is done by queue and DFS by stack is there any difference between the two?

The difference between DFS and BFS is similar to this question. In DFS we will find each path possible in the tree one by one. It will perform all steps for a path first whereas BFS will build all paths together one step at a time.

So, a queue would work perfectly for this question. The only difference between the two algorithms using queue and stack will be the way in which they are formed. Stack will form all strings completely one by one whereas the queue will form all the strings together i.e. after x number of passes all the strings will have a length of x.

**Below is the implementation of the above approach:**

|  |
| --- |
| // Java implementation of the approach  import java.io.\*;  import java.util.\*;    class GFG  {      // Function to return a vector that contains      // all the generated letter combinations      static ArrayList<String> letterCombinationsUtil(int[] number, int n,                                                              String[] table)      {              // To store the generated letter combinations              ArrayList<String> list = new ArrayList<>();                Queue<String> q = new LinkedList<>();              q.add("");                while(!q.isEmpty())              {                      String s = q.remove();                        // If complete word is generated                      // push it in the list                      if (s.length() == n)                      list.add(s);                      else                      {                          String val = table[number[s.length()]];                          for (int i = 0; i < val.length(); i++)                          {                              q.add(s + val.charAt(i));                          }                      }              }              return list;      }        // Function that creates the mapping and      // calls letterCombinationsUtil      static void letterCombinations(int[] number, int n)      {              // table[i] stores all characters that              // corresponds to ith digit in phone              String[] table = { "", "", "abc", "def", "ghi", "jkl",              "mno", "pqrs", "tuv", "wxyz" };                ArrayList<String> list =                          letterCombinationsUtil(number, n, table);                // Print the contents of the list              for (int i = 0; i < list.size(); i++)              {                  System.out.print(list.get(i) + " ");              }      }        // Driver code      public static void main(String args[])      {              int[] number = { 2, 3 };              int n = number.length;              letterCombinations(number, n);      }  }    // This code is contributed by rachana soma |

**Output:**

ad ae af bd be bf cd ce cf