原始问题如下：手机上面的数字键均对应了几个字符，譬如 2 对应了 a,b,c。问题是当输入一段数字后，求出所有可能的字符组合，（可以想象一下发短信时候的状况，每当按几个数字键后，均给出可能的汉语拼音，当然这个要求就更高了，本题只要求给出所有可能的组合）。举个例子输入 4，2 键后，则给出 GA,GB,GC,HA,HB,HC,IA,IB,IC 组合。
当然对于大多数人来说就是几层循环就搞定了，每层遍历，内部输出。但是我们当然是追求更高的算法了。若按普通做法，那么每多按一个数字就会导致再添加一个 for 循环了，明显不合适。下面给出让你佩服的方法：

[cpp] view plaincopy

```cpp
#include<iostream>
using namespace std;

const int MaxLength = 9;
char c[10][10] = {"", "", "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TUV", "WXYZ"};
int total[10] = {0,0,3,3,3,3,3,4,3,4};

int main()
{
    int number[MaxLength] = {2,3,4}; //本例输入数字 2,3,4
    int answer[MaxLength] = {0};
    int len = 3;

    while(true){
        for(int i = 0; i < len; i++)
            printf("%c", c[number[i]][answer[i]]);
        printf("\n");

        int k = len - 1;
        while(k >= 0){
            if(answer[k] < total[number[k]] - 1){
                answer[k]++;
                break;
            }
            else{
                answer[k] = 0;
                k--;
            }
        }
        if(k < 0)
            break;
    }
    return 0;
}
```

居然只有两层循环就搞定了，读了半天都没读懂，不过读懂过后令人非常钦佩啊，原来还可以这么写！程序可读性不强，不过也说明咱读代码水平不高，话说牛叉的算法不都是精简而高效，可读性不强么。
感叹之余，观察此算法也是非常适合排列组合的生成么。厉害啊。

另外还有递归实现的，原理和数据结构类似。

其实它的实质就是遍历一个多叉树，因此还有使用前中后序遍历算法来解决这种题目的方案：
摘自《编程之美：微软技术面试心得》上的"电话号码对应英语单词"问题

电话的号码盘一般可以用于输入字母，如用 2 可以输入 A、B、C，用 3 可以输入 D、E、F 等。如对于号码 5869872，可以一次输出其代表的所有字母组合，如：JTMWTPA、JTMWTPB……
1、您是否可以根据这样的对应关系设计一个程序，尽可能的从这些字母组合中找到一个有意义的单词 来表达一个电话号码呢？如：可以用单词"computer"来描述号码 26678837。
2、对于一个电话号码，是否可以用一个单词来代表呢？怎样才是最快的方法呢？显然，肯定不是所有的电话号码都能够对应到单词上去。但是根据问题 1 的解答，思路相对就会比较清晰。

```
#include <iostream>
#include <vector>
#include <string>


std::vector<std::string> c(10);

std::vector<int> total(10);



void RecursiveSearch(std::vector<int>& number, std::vector<int>& answer, in
t index, int n)
{
    if (index == n)
    {
        for (int i = 0; i < n; i++)
            printf("%c", c[number[i]][answer[i]]);
        printf("\n");
        return;
    }
    for (answer[index] = 0;
```

```
        answer[index] < total[number[index]];
        answer[index]++)
    {
        RecursiveSearch(number, answer, index + 1, n);
    }
}


void DirectSearch_i(std::vector<int>& number)
{
    int telLength = number.size();

    int resNum = 1;
    for (int i = 0; i < telLength; ++i)
    {
        if (total[number[i]] > 0)
        {
            resNum *= total[number[i]];
        }
    }

    std::vector<std::string> result(resNum, "");


    for (int i = 0; i < telLength; ++i)
    {
        if (0 == total[number[i]])
        {
            continue;
        }

        int loopNum = resNum / total[number[i]];
        int div = 1;
        for (int m = i + 1; m < telLength; ++m)
        {
            if (total[number[m]] > 0)
            {
                div *= total[number[m]];
            }
        }

        int multi = resNum / div;
```

```
        int m = -1;
        int n = -1;
        for (int j = 0; j < multi; ++j)
        {
            n++;
            n = n % (total[number[i]]);

            for (int k = 0; k < div; ++k)
            {
                m++;

                result[m].push_back(c[number[i]][n]);
            }
        }
    }


    for (int i = 0; i < resNum; ++i)
    {
        std::cout << result[i] << std::endl;
    }
}


void DirectSearch(std::vector<int>& number, std::vector<int>& answer)
{
    int telLength = number.size();

    while (true)
    {
        for (int i = 0; i < telLength; i++)
            printf("%c", c[number[i]][answer[i]]);
        printf("\n");
        int k = telLength - 1;
        while (k >= 0)
        {
            if (answer[k] < total[number[k]] - 1)
            {
                answer[k]++;
                break;
            }
            else
            {
                answer[k] = 0; k--;
```

```
        }
      }
      if (k < 0)
        break;
    }
}


int main(void)
{
    c[0] = "";          // 0
    c[1] = "";          // 1
    c[2] = "ABC";       // 2
    c[3] = "DEF";       // 3
    c[4] = "GHI";       // 4
    c[5] = "JKL";       // 5
    c[6] = "MNO";       // 6
    c[7] = "PQRS";      // 7
    c[8] = "TUV";       // 8
    c[9] = "WXYZ";      // 9


    total[0] = 0;
    total[1] = 0;
    total[2] = 3;
    total[3] = 3;
    total[4] = 3;
    total[5] = 3;
    total[6] = 3;
    total[7] = 4;
    total[8] = 3;
    total[9] = 4;


    std::vector<int> number;
    //number.push_back(1);
    number.push_back(2);
    number.push_back(3);
    //number.push_back(4);
    //number.push_back(5);
    //number.push_back(6);
    //number.push_back(7);
```

```
    int telLength = number.size();


    std::vector<int> answer(telLength, 0); // 开始取的都是数字上的第一个字符


    // Method 1:
    std::cout << "Use method 1:" << std::endl;
    DirectSearch_i(number);

    //Method 2:
    std::cout << "Use method 2:" << std::endl;
    DirectSearch(number, answer);

    // Method 3:
    std::cout << "Use method 3:" << std::endl;
    RecursiveSearch(number, answer, 0, telLength);


    system("pause");

    return 0;
}
```

说明：
1，原文的数据结构都采用数组的方法，我依据《Effective C++》上说法“所有的数组几乎都可用string和vector等template替换”，将它们都替换成string和vector，看起来更加C++。
2，Method 1 是我自己在看它的答案之前自己想出来的，看起来比较消耗内存。
3，关于它提供的两种解法，碰到数字为“0”或“1”时会出现问题。
4，关于它提供的两种解法，思路上更加抽象，抽象到了树形结构这一层，采用这种思路能够解决很多其它类似的问题。
5，个人更加偏向于直接解法（Method 2），递归会把堆栈拉得很长，数据量大了可能会出问题。
6，关于 Method 2，有点像做加法，每个位有自己的进制值（total[number[k]]），每次加 1（从最低位开始），当满了时则向高位进 1，直到溢出（最高位的 index 为 0，当 index<0 时退出）。
7，关于 Method 3，很像遍历一颗树，采用的是前序法，使用 index 标记遍历到了哪一层，当 index==n 时（depth=2），即表示到了最深处，可以开始打印输出了。