Answer of 1

```
import java.util.ArrayList;
import java.util.List;

class KnapsackSolution {

    static class Item {
        int weight;
        int value;

        Item(int weight, int value) {
            this.weight = weight;
            this.value = value;
        }
    }

     public List<List<Item>> knapsack(int[] weights, int[]
values, int W, int V) {
        List<Item> items = new ArrayList<>();
        for (int i = 0; i < weights.length; i++) {
            items.add(new Item(weights[i], values[i]));
        }

        List<List<Item>> result = new ArrayList<>();
        findSubsets(items, W, V, new ArrayList<>(), 0, 0, 0,
result);
        return result;
    }

    public void findSubsets(List<Item> items, int W, int V,
List<Item> currentSubset, int currentWeight, int currentValue,
int index, List<List<Item>> result) {
        // base case
        if (index == items.size()) {
            if (currentWeight <= W && currentValue >= V) {
                result.add(new ArrayList<>(currentSubset));
            }
            return;
        }

        findSubsets(items, W, V, currentSubset, currentWeight,
currentValue, index + 1, result);

        // check if current item satisifes the constraint
        if (currentWeight + items.get(index).weight <= W) {
            currentSubset.add(items.get(index));

            findSubsets(items, W, V, currentSubset,
currentWeight + items.get(index).weight, currentValue +
items.get(index).value, index + 1, result);

            currentSubset.remove(currentSubset.size() - 1);
        }
    }
}
```

Answer of 2

```
class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        List<Integer> current = new ArrayList<>();
        boolean[] used = new boolean[nums.length];

        backtrack(result, current, used, nums);

        return result;
    }

    private void backtrack(List<List<Integer>> result,
List<Integer> current, boolean[] used, int[] nums) {
        // base case
        if (current.size() == nums.length) {
            result.add(new ArrayList<>(current));
            return;
        }

        for (int i = 0; i < nums.length; i++) {
            // skip if already used
            if (used[i]) {
                continue;
            }


            current.add(nums[i]);
            used[i] = true;

            // recurse to the next options
            backtrack(result, current, used, nums);

            // backtrack remove the last element and mark it as
unused
            current.remove(current.size() - 1);
            used[i] = false;
        }
    }
}
```

Answer of 3

```
class Solution {

    // digit-letter mapping
    private static final String[] KEYPAD = {
        "",     // 0 - unused
        "",     // 1 - unused
        "abc",  // 2
        "def",  // 3
        "ghi",  // 4
        "jkl",  // 5
        "mno",  // 6
        "pqrs", // 7
        "tuv",  // 8
        "wxyz"  // 9
    };

    public List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<>();

        if (digits == null || digits.length() == 0) {
            return result;
        }

        backtrack(result, new StringBuilder(), digits, 0);

        return result;
    }

    private void backtrack(List<String> result, StringBuilder
current, String digits, int index) {
        // base case
        if (index == digits.length()) {
            result.add(current.toString());
            return;
        }

        // get the letter
        // c - '0' as c is digit
        String letters = KEYPAD[digits.charAt(index) - '0'];

        // go through each letter in array
        for (char letter : letters.toCharArray()) {
            // add to current list set
            current.append(letter);

            // backtrack to next node on left
            backtrack(result, current, digits, index + 1);

            // remove from current
            current.deleteCharAt(current.length() - 1);
        }
    }
}
```