136. Palindrome Partitioning

Description | Notes | Testcase | Judge

Given a string *s*, partition *s* such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

Have you met this question in a real interview? Yes

Example

Given s = `"aab"`, return:

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

Tags

```
// version 1: shorter but slower
public class Solution {
```

```java
    /**
     * @param s: A string
     * @return: A list of lists of string
     */
    public List<List<String>> partition(String s) {
        List<List<String>> results = new ArrayList<>();
        if (s == null || s.length() == 0) {
            return results;
        }

        List<String> partition = new ArrayList<String>();
        helper(s, 0, partition, results);

        return results;
    }

    private void helper(String s,
                        int startIndex,
                        List<String> partition,
                        List<List<String>> results) {
        if (startIndex == s.length()) {
            results.add(new ArrayList<String>(partition));
            return;
        }

        for (int i = startIndex; i < s.length(); i++) {
            String subString = s.substring(startIndex, i + 1);
            if (!isPalindrome(subString)) {
                continue;
            }
            partition.add(subString);
            helper(s, i + 1, partition, results);
            partition.remove(partition.size() - 1);
        }
    }

    private boolean isPalindrome(String s) {
        for (int i = 0, j = s.length() - 1; i < j; i++, j--) {
            if (s.charAt(i) != s.charAt(j)) {
                return false;
            }
        }
        return true;
    }
}
```

153. Combination Sum II ☆

Description | Notes | >_ Testcase | Judge

Given a collection of candidate numbers (*C*) and a target number (*T*), find all unique combinations in *C* where the candidate numbers sums to *T*.

Each number in *C* may only be used once in the combination.

Notice

- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
- The solution set must not contain duplicate combinations.

Have you met this question in a real interview? Yes

Example

Given candidate set `[10,1,6,7,2,1,5]` and target `8`,

A solution set is:

Example

Given candidate set `[10,1,6,7,2,1,5]` and target `8`,

A solution set is:

```
[
  [1,7],
  [1,2,5],
  [2,6],
  [1,1,6]
]
```

```java
public class Solution {
    /**
     * @param num: Given the candidate numbers
     * @param target: Given the target number
     * @return: All the combinations that sum to target
     */
    public List<List<Integer>> combinationSum2(int[] candidates,
            int target) {
        List<List<Integer>> results = new ArrayList<>();
        if (candidates == null || candidates.length == 0) {
            return results;
        }

        Arrays.sort(candidates);
        List<Integer> combination = new ArrayList<Integer>();
        helper(candidates, 0, combination, target, results);

        return results;
    }

    private void helper(int[] candidates,
                        int startIndex,
                        List<Integer> combination,
                        int target,
                        List<List<Integer>> results) {
        if (target == 0) {
            results.add(new ArrayList<Integer>(combination));
            return;
        }

        for (int i = startIndex; i < candidates.length; i++) {
            if (i != startIndex && candidates[i] == candidates[i - 1]) {
                continue;
            }
            if (target < candidates[i]) {
                break;
            }
            combination.add(candidates[i]);
            helper(candidates, i + 1, combination, target - candidates[i], results);
            combination.remove(combination.size() - 1);
        }
    }
}
```

135. Combination Sum

Description | Notes | Testcase | Judge

Given a set of candidate numbers (***C***) and a target number (***T***), find all unique combinations in ***C*** where the candidate numbers sums to ***T***.

The **same** repeated number may be chosen from ***C*** unlimited number of times.

Notice

- All numbers (including target) will be positive integers.
- Elements in a combination (a_1, a_2, … , a_k) must be in non-descending order. (ie, $a_1 \le a_2 \le \ldots \le a_k$).
- The solution set must not contain duplicate combinations.

Have you met this question in a real interview? Yes

Example

Given candidate set `[2,3,6,7]` and target `7`, a solution set is:

```
[7]
[2, 2, 3]
```

```
// version 1: Remove duplicates & generate a new array
public class Solution {
    /**
     * @param candidates: A list of integers
     * @param target:An integer
     * @return: A list of lists of integers
     */
    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> results = new ArrayList<>();
        if (candidates == null || candidates.length == 0) {
            return results;
        }

        int[] nums = removeDuplicates(candidates);

        dfs(nums, 0, new ArrayList<Integer>(), target, results);

        return results;
    }

    private int[] removeDuplicates(int[] candidates) {
        Arrays.sort(candidates);

        int index = 0;
        for (int i = 0; i < candidates.length; i++) {
            if (candidates[i] != candidates[index]) {
                candidates[++index] = candidates[i];
            }
        }

        int[] nums = new int[index + 1];
        for (int i = 0; i < index + 1; i++) {
            nums[i] = candidates[i];
        }

        return nums;
    }

    private void dfs(int[] nums,
                     int startIndex,
                     List<Integer> combination,
                     int remainTarget,
                     List<List<Integer>> results) {
        if (remainTarget == 0) {
            results.add(new ArrayList<Integer>(combination));
```

```
            return;
        }

        for (int i = startIndex; i < nums.length; i++) {
            if (remainTarget < nums[i]) {
                break;
            }
            combination.add(nums[i]);
            dfs(nums, i, combination, remainTarget - nums[i], results);
            combination.remove(combination.size() - 1);
        }
    }
}

// version 2: reuse candidates array
public class Solution {
    public  List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        if (candidates == null) {
            return result;
        }

        List<Integer> combination = new ArrayList<>();
        Arrays.sort(candidates);
        helper(candidates, 0, target, combination, result);

        return result;
    }

     void helper(int[] candidates,
                 int index,
                 int target,
                 List<Integer> combination,
                 List<List<Integer>> result) {
        if (target == 0) {
            result.add(new ArrayList<Integer>(combination));
            return;
        }

        for (int i = index; i < candidates.length; i++) {
            if (candidates[i] > target) {
                break;
            }

            if (i != index && candidates[i] == candidates[i - 1]) {
                continue;
            }

            combination.add(candidates[i]);
```

```
            helper(candidates, i, target - candidates[i], combination, result);
            combination.remove(combination.size() - 1);
        }
    }
}
```

18. Subsets II

Description | Notes | Testcase | Judge

Given a list of numbers that may has duplicate numbers, return all possible subsets

Notice

- Each element in a subset must be in *non-descending* order.
- The ordering between two subsets is free.
- The solution set must not contain duplicate subsets.

Have you met this question in a real interview? Yes

Example

If S = `[1,2,2]`, a solution is:

```
[
  [2],
  [1],
  [1,2,2],
  [2,2],
  [1,2],
  []
]
```

```
// return List<List<Integer>>
class Solution {
    /**
     * @param nums: A set of numbers.
     * @return: A list of lists. All valid subsets.
     */
    public List<List<Integer>> subsetsWithDup(int[] nums) {
        // write your code here
        List<List<Integer>> results = new ArrayList<List<Integer>>();
        if (nums == null) return results;

        if (nums.length == 0) {
            results.add(new ArrayList<Integer>());
            return results;
        }
        Arrays.sort(nums);

        List<Integer> subset = new ArrayList<Integer>();
        helper(nums, 0, subset, results);

        return results;


    }
    public void helper(int[] nums, int startIndex, List<Integer> subset, List<List<Integer>> results){
        results.add(new ArrayList<Integer>(subset));
        for(int i=startIndex; i<nums.length; i++){
            if (i != startIndex && nums[i]==nums[i-1]) {
                continue;
            }
            subset.add(nums[i]);
            helper(nums, i+1, subset, results);
            subset.remove(subset.size()-1);
        }
    }
}
```

16. Permutations II

Description | Notes | Testcase | Judge

Given a list of numbers with duplicate number in it. Find all **unique** permutations.

Have you met this question in a real interview? Yes

Example

For numbers `[1,2,2]` the unique permutations are:

```
[
  [1,2,2],
  [2,1,2],
  [2,2,1]
]
```

```
class Solution {
    /**
     * @param nums: A list of integers.
     * @return: A list of unique permutations.
     */
    public List<List<Integer>> permuteUnique(int[] nums) {

        ArrayList<List<Integer>> results = new ArrayList<List<Integer>>();

        if (nums == null) {
            return results;
        }

        if(nums.length == 0) {
            results.add(new ArrayList<Integer>());
            return results;
        }

        Arrays.sort(nums);
        ArrayList<Integer> list = new ArrayList<Integer>();
        int[] visited = new int[nums.length];
        for ( int i = 0; i < visited.length; i++){
            visited[i] = 0;
        }

        helper(results, list, visited, nums);
        return results;
    }


    public void helper(ArrayList<List<Integer>> results,
                ArrayList<Integer> list, int[] visited, int[] nums) {

        if(list.size() == nums.length) {
            results.add(new ArrayList<Integer>(list));
            return;
        }

        for(int i = 0; i < nums.length; i++) {
            if ( visited[i] == 1 || ( i != 0 && nums[i] == nums[i - 1]
            && visited[i-1] == 0)){
                continue;
            }
            /*
            上面的判断主要是为了去除重复元素影响。
            比如，给出一个排好序的数组，[1,2,2]，那么第一个2和第二2如果在结果中互换位置，
```

```
        我们也认为是同一种方案，所以我们强制要求相同的数字，原来排在前面的，在结果
        当中也应该排在前面，这样就保证了唯一性。所以当前面的2还没有使用的时候，就
        不应该让后面的2使用。
        */
        visited[i] = 1;
        list.add(nums[i]);
        helper(results, list, visited, nums);
        list.remove(list.size() - 1);
        visited[i] = 0;
    }
  }
}
```

15. Permutations

Description | Notes | Testcase | Judge

Given a list of numbers, return all possible permutations.

Notice

You can assume that there is no duplicate numbers in the list.

Have you met this question in a real interview? Yes

Example

For nums = `[1,2,3]`, the permutations are:

```
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

Challenge

```java
public class Solution {
    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> results = new ArrayList<>();
        if (nums == null) {
            return results;
        }

        if (nums.length == 0) {
            results.add(new ArrayList<Integer>());
            return results;
        }

        List<Integer> permutation = new ArrayList<Integer>();
        Set<Integer> set = new HashSet<>();
        helper(nums, permutation, set, results);

        return results;
    }

    // 1. 找到所有以permutation 开头的排列
    public void helper(int[] nums,
                       List<Integer> permutation,
                       Set<Integer> set,
                       List<List<Integer>> results) {
        // 3. 递归的出口
        if (permutation.size() == nums.length) {
            results.add(new ArrayList<Integer>(permutation));
            return;
        }


        // [3] => [3,1], [3,2], [3,4] ...
        for (int i = 0; i < nums.length; i++) {
            if (set.contains(nums[i])) {
                continue;
            }

            permutation.add(nums[i]);
            set.add(nums[i]);
            helper(nums, permutation, set, results);
            set.remove(nums[i]);
            permutation.remove(permutation.size() - 1);
        }

    }
}
```

121. Word Ladder II

Description | Notes | Testcase | Judge

Given two words (*start* and *end*), and a dictionary, find all shortest transformation sequence(s) from *start* to *end*, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the dictionary

Notice

- All words have the same length.
- All words contain only lowercase alphabetic characters.

Have you met this question in a real interview? Yes

Example

Given:
start = `"hit"`
end = `"cog"`
dict = `["hot","dot","dog","lot","log"]`

Return

```
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
```

Tags

给出两个单词（start和end）和一个字典，找出所有从start到end的最短转换序列

比如：

每次只能改变一个字母。
变换过程中的中间单词必须在字典中出现

```
public class Solution {
    public List<List<String>> findLadders(String start, String end,
            Set<String> dict) {
        List<List<String>> ladders = new ArrayList<List<String>>();
        Map<String, List<String>> map = new HashMap<String, List<String>>();
        Map<String, Integer> distance = new HashMap<String, Integer>();

        dict.add(start);
        dict.add(end);

        bfs(map, distance, start, end, dict);

        List<String> path = new ArrayList<String>();

        dfs(ladders, path, end, start, distance, map);

        return ladders;
    }

    void dfs(List<List<String>> ladders, List<String> path, String crt,
            String start, Map<String, Integer> distance,
            Map<String, List<String>> map) {
        path.add(crt);
        if (crt.equals(start)) {
            Collections.reverse(path);
            ladders.add(new ArrayList<String>(path));
            Collections.reverse(path);
        } else {
            for (String next : map.get(crt)) {
                if (distance.containsKey(next) && distance.get(crt) == distance.get(next) + 1) {
                    dfs(ladders, path, next, start, distance, map);
                }
            }
        }
        path.remove(path.size() - 1);
    }

    void bfs(Map<String, List<String>> map, Map<String, Integer> distance,
            String start, String end, Set<String> dict) {
```

```
        Queue<String> q = new LinkedList<String>();
        q.offer(start);
        distance.put(start, 0);
        for (String s : dict) {
            map.put(s, new ArrayList<String>());
        }

        while (!q.isEmpty()) {
            String crt = q.poll();

            List<String> nextList = expand(crt, dict);
            for (String next : nextList) {
                map.get(next).add(crt);
                if (!distance.containsKey(next)) {
                    distance.put(next, distance.get(crt) + 1);
                    q.offer(next);
                }
            }
        }
    }

    List<String> expand(String crt, Set<String> dict) {
        List<String> expansion = new ArrayList<String>();

        for (int i = 0; i < crt.length(); i++) {
            for (char ch = 'a'; ch <= 'z'; ch++) {
                if (ch != crt.charAt(i)) {
                    String expanded = crt.substring(0, i) + ch
                            + crt.substring(i + 1);
                    if (dict.contains(expanded)) {
                        expansion.add(expanded);
                    }
                }
            }
        }

        return expansion;
    }
}
```