第 1 章
Backtracking

1.1 Regular Expression Matching

Description

Implement regular expression matching with support for `'.'` and `'*'`.

`'.'` Matches any single character. `'*'` Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Some examples:

```
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "a*") → true
isMatch("aa", ".*") → true
isMatch("ab", ".*") → true
isMatch("aab", "c*a*b") → true
```

Solution

```
public boolean isMatch(String s, String p) {
    if (p.isEmpty()) {
        return s.isEmpty();
    } else if (p.length() == 1) {
        return s.length() == 1 && isEqual(s, p);
    } else if (p.charAt(1) != '*') {
        return s.length() > 0 && isEqual(s, p) && isMatch(s.substring(1), p.substring(1));
    } else {
        if (s.length() > 0 && isEqual(s, p)) {
            return isMatch(s, p.substring(2)) || isMatch(s.substring(1), p);
        } else {
            return isMatch(s, p.substring(2));
        }
    }
}

private boolean isEqual(String s, String p) {
    return s.charAt(0) == p.charAt(0) || p.charAt(0) == '.';
}
```

1.2 Wildcard Matching

Description

Implement wildcard pattern matching with support for '?' and '*'.

'?' Matches any single character. '*' Matches any sequence of characters (including the empty sequence).

```
Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

Solution

```
public boolean isMatch2(String s, String p) {
    int is = 0, ip = 0, ks = -1, kp = -1;

    while (is < s.length()) {
        if (ip < p.length() && (s.charAt(is) == p.charAt(ip) || p.charAt(ip) == '?')) {
            is++;
            ip++;
        } else if (ip < p.length() && p.charAt(ip) == '*') {
            ks = is;
            kp = ip;
            ip++;
        } else if (kp != -1) {
            is = ++ks;
            ip = kp + 1;
        } else {
            return false;
        }
    }

    for ( ; ip < p.length() && p.charAt(ip) == '*'; ip++);
    return ip == p.length();
}
```

1.3 Letter Combinations of a Phone Number

Description

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.

图 1-1 Phone Keyboard

Input: Digit string `"23"`

Output: `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`.

Solution I

```
private final String[] ARR = {
        "", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"
};

public List<String> letterCombinations(String digits) {
    List<String> list = new LinkedList<>();
    if (!digits.isEmpty()) {
        helper(digits, 0, list, "");
    }
    return list;
}

private void helper(String digits, int start, List<String> list, String s) {
    if (start >= digits.length()) {
        list.add(s);
        return;
    }
    int n = digits.charAt(start) - '0';
    for (char c : ARR[n].toCharArray()) {
        helper(digits, start + 1, list, s + c);
    }
}
```

Solution II

```
public List<String> letterCombinations(String digits) {
    LinkedList<String> queue = new LinkedList<String>();
    if (digits.length() == 0) {
        return queue;
    }

    Queue<String> next = new LinkedList<>();
    queue.add("");

    for (int i = 0; i < digits.length() && !queue.isEmpty(); ) {
        String s = queue.poll();
        int n = digits.charAt(i) - '0';
        for (char c : ARR[n].toCharArray()) {
            next.add(s + c);
        }
        if (queue.isEmpty()) {
            queue.addAll(next);
            next.clear();
            i++;
        }
    }
    return queue;
}
```

1.4 Generate Parentheses

Description

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given `n = 3`, a solution set is:

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

Solution

```
public List<String> generateParenthesis(int n) {
    List<String> result = new LinkedList<String>();
    dfs(result, n, "", 0, 0);
    return result;
}

private void dfs(List<String> result, int n, String str, int left, int right) {
    if (left == n && right == n) {
        result.add(str);
        return;
    }
    if (left > n || right > n || left < right) {
        return;
    }
    dfs(result, n, str + "(", left + 1, right);
    dfs(result, n, str + ")", left, right + 1);
}
```

1.5 Permutations

Description

Given a collection of distinct numbers, return all possible permutations.

For example, `[1,2,3]` have the following permutations:

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

Solution

```
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    permute(nums, result, 0);
    return result;
}

public void permute(int[] nums, List<List<Integer>> result, int start) {
    if (start >= nums.length) {
        List<Integer> list = new ArrayList<Integer>();
        for (Integer n : nums) {
            list.add(n);
        }
        result.add(list);
    }

    for (int i = start; i < nums.length; i++) {
        swap(nums, start, i);
        permute(nums, result, start + 1);
        swap(nums, start, i);
    }
}

public static void swap(int[] nums, int left, int right) {
    int temp = nums[left];
    nums[left] = nums[right];
    nums[right] = temp;
}
```

1.6 Word Search

Description

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where adjacent cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example, Given board =

```
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]
```

word = `"ABCCED"`, -> returns true,

word = `"SEE"`, -> returns true,

word = `"ABCB"`, -> returns false.

Solution

```
public boolean exist(char[][] board, String word) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (dfs(board, i, j, word, 0)) {
                return true;
            }
        }
    }
    return false;
}

private boolean dfs(char[][] board, int i, int j, String word, int start) {
    if (start == word.length()) {
        return true;
    }
    if (i < 0 || i >= board.length || j < 0 || j >= board[0].length) {
        return false;
    }
    if (board[i][j] != word.charAt(start)) {
        return false;
    }

    board[i][j] ^= '#';
    boolean flag = dfs(board, i + 1, j, word, start + 1)
            || dfs(board, i - 1, j, word, start + 1)
            || dfs(board, i, j + 1, word, start + 1)
            || dfs(board, i, j - 1, word, start + 1);
    board[i][j] ^= '#';
    return flag;
}
```

1.7 Word Search II

Description

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where adjacent cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example, Given words = `["oath","pea","eat","rain"]` and board =

```
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]
```

Return `["eat","oath"]`.

Note: You may assume that all inputs are consist of lowercase letters a-z.

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

Solution

```
private class Trie {
    Trie[] nodes = new Trie[26];
    String word;
}

private void buildTrie(Trie trie, String word) {
    for (int i = 0; i < word.length(); i++) {
        if (trie.nodes[word.charAt(i) - 'a'] == null) {
            trie.nodes[word.charAt(i) - 'a'] = new Trie();
        }
        trie = trie.nodes[word.charAt(i) - 'a'];
    }
    trie.word = word;
}
```

```
public List<String> findWords(char[][] board, String[] words) {
    Trie trie = new Trie();
    for (String word : words) {
        buildTrie(trie, word);
    }
    Set<String> set = new HashSet<String>();
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            dfs(set, board, i, j, trie);
        }
    }
    return new LinkedList<String>(set);
}

private void dfs(Set<String> set, char[][] board, int i, int j, Trie trie) {
    if (i < 0 || i >= board.length || j < 0 || j >= board[0].length) {
        return;
    }
    if (trie == null) {
        return;
    }
    char c = board[i][j];
    if (c < 'a' || c > 'z') {
        return;
    }
    trie = trie.nodes[c - 'a'];
    if (trie == null) {
        return;
    }
    if (trie.word != null) {
        set.add(trie.word);
    }

    board[i][j] ^= '#';
    dfs(set, board, i + 1, j, trie);
    dfs(set, board, i - 1, j, trie);
    dfs(set, board, i, j + 1, trie);
    dfs(set, board, i, j - 1, trie);
    board[i][j] ^= '#';
}
```

1.8 Word Break

Description

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, determine if s can be segmented into a space-separated sequence of one or more dictionary words. You may assume the dictionary does not contain duplicate words.

For example, given s = `"leetcode"`, dict = `["leet", "code"]`.

Return true because `"leetcode"` can be segmented as `"leet code"`.

Solution

```
public boolean wordBreak(String s, List<String> wordDict) {
    int n = s.length();

    boolean[] dp = new boolean[n + 1];
    dp[0] = true;

    for (int i = 1; i <= s.length(); i++) {
        for (String word : wordDict) {
            int j = i - word.length();
            if (j >= 0 && dp[j] && s.substring(j, i).equals(word)) {
                dp[i] = true;
                break;
            }
        }
    }

    return dp[n];
}
```

1.9 Word Break II

Description

Given a non-empty string s and a dictionary wordDict containing a list of non-empty words, add spaces in s to construct a sentence where each word is a valid dictionary word. You may assume the dictionary does not contain duplicate words.

Return all such possible sentences.

For example, given s = `"catsanddog"`, dict = `["cat", "cats", "and", "sand", "dog"]`.

A solution is `["cats and dog", "cat sand dog"]`.

Solution

```
public List<String> wordBreak(String s, List<String> wordDict) {
    HashMap<String, List<String>> cache = new HashMap<>();
    cache.put("", Arrays.asList(""));
    return dfs(s, new HashSet<String>(wordDict), cache);
}

private List<String> dfs(String s, HashSet<String> wordDict, HashMap<String, List<String>> cac
    if (cache.containsKey(s)) {
        return cache.get(s);
    }
    List<String> result = new LinkedList<>();
    for (int i = 0; i < s.length(); i++) {
        String t = s.substring(i);
        if (wordDict.contains(t)) {
            List<String> list = dfs(s.substring(0, i), wordDict, cache);
            if (list != null) {
                for (String ss : list) {
                    result.add((ss.length() > 0 ? ss + " " : "") + t);
                }
            }
        }
    }
    cache.put(s, result);
    return result;
}
```

1.10 Combination Sum

Description

Given a set of candidate numbers (C) (without duplicates) 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.

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

For example, given candidate set `[2, 3, 6, 7]` and target 7,

A solution set is: `[` [7], [2, 2, 3]]

Solution I

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new LinkedList<>();
    helper(candidates, result, new LinkedList<>(), target, 0);
    return result;
}

private void helper(int[] candidates, List<List<Integer>> result, List<Integer> list, int targ
    if (target < 0) {
        return;
    } else if (target == 0) {
        result.add(new LinkedList<>(list));
        return;
    } else if (index >= candidates.length) {
        return;
    }

    list.add(candidates[index]);
    helper(candidates, result, list, target - candidates[index], index);
    list.remove(list.size() - 1);

    helper(candidates, result, list, target, index + 1);
}
```

Solution II

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new LinkedList<>();
    dfs(candidates, target, 0, result, new LinkedList<Integer>());
    return result;
}

private void dfs(int[] candidates, int target, int start, List<List<Integer>> result, List<Int
    if (target < 0) {
        return;
    }
    if (target == 0) {
        result.add(new LinkedList<>(list));
        return;
    }
    for (int i = start; i < candidates.length; i++) {
        list.add(candidates[i]);
        dfs(candidates, target - candidates[i], i, result, list);
        list.remove(list.size() - 1);
    }
}
```

1.11 Combination Sum II

Description

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.

Note:

All numbers (including target) will be positive integers.

The solution set must not contain duplicate combinations.

For example, given candidate set `[10, 1, 2, 7, 6, 1, 5]` and target 8,

A solution set is:

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

Solution I

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new LinkedList<>();
    Arrays.sort(candidates);
    helper(candidates, result, new LinkedList<>(), target, 0);
    return result;
}

private void helper(int[] candidates, List<List<Integer>> result, List<Integer> list, int targ
    if (target < 0) {
        return;
    } else if (target == 0) {
        result.add(new LinkedList<>(list));
        return;
    } else if (index >= candidates.length) {
        return;
    }

    list.add(candidates[index]);
    helper(candidates, result, list, target - candidates[index], index + 1);
    list.remove(list.size() - 1);

    for ( ; index < candidates.length - 1 && candidates[index + 1] == candidates[index]; index

    helper(candidates, result, list, target, index + 1);
}
```

Solution II

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new LinkedList<List<Integer>>();
    Arrays.sort(candidates);
    dfs(candidates, 0, target, result, new LinkedList<Integer>());
    return result;
}

private void dfs(int[] candidates, int start, int target, List<List<Integer>> result, List<Int
    if (target < 0) {
        return;
    }

    if (target == 0) {
        result.add(new LinkedList<Integer>(path));
        return;
    }

    for (int i = start; i < candidates.length; i++) {
        if (i > start && candidates[i] == candidates[i - 1]) {
            continue;
        }

        path.add(candidates[i]);

        // 关键是这里变成 i + 1
        dfs(candidates, i + 1, target - candidates[i], result, path);
        path.remove(path.size() - 1);
    }
}
```

1.12 Combination Sum III

Description

Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Example 1:

Input: `k = 3, n = 7`

Output: `[[1,2,4]]`

Example 2:

Input: `k = 3, n = 9`

Output: `[[1,2,6], [1,3,5], [2,3,4]]`

Solution I

```
public List<List<Integer>> combinationSum(int k, int n) {
    List<List<Integer>> result = new LinkedList<>();
    helper(k, n, result, new LinkedList<>(), 1);
    return result;
}

private void helper(int k, int n, List<List<Integer>> result, List<Integer> list, int cur) {
    if (n == 0 && k == 0) {
        result.add(new LinkedList<>(list));
        return;
    }

    if (cur > 9) {
        return;
    }

    list.add(cur);
    helper(k - 1, n - cur, result, list, cur + 1);
    list.remove(list.size() - 1);

    helper(k, n, result, list, cur + 1);
}
```

Solution II

```
public List<List<Integer>> combinationSum(int k, int n) {
    List<List<Integer>> result = new LinkedList<>();
    dfs(n, k, 1, result, new LinkedList<Integer>());
    return result;
}

private void dfs(int target, int k, int start, List<List<Integer>> result, List<Integer> list)
    if (target == 0 && k == 0) {
        result.add(new LinkedList<>(list));
        return;
    }
    if (target <= 0 || k <= 0) {
        return;
    }
    for (int i = start; i <= 9; i++) {
        list.add(i);
        dfs(target - i, k - 1, i + 1, result, list);
        list.remove(list.size() - 1);
    }
}
```