,

目录

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 Permutations II

Description

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

`[1,1,2]` have the following unique permutations:

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

Solution

1.7 Permutation Sequence

Description

The set `[1,2,3,⋯,n]` contains a total of `n!` unique permutations.

By listing and labeling all of the permutations in order, We get the following sequence (ie, for n = 3):

```
"123"
"132"
"213"
"231"
"312"
"321"
```

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

Solution

```
public String getPermutation(int n, int k) {
    int[] nums = new int[n];

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

    for (int i = 1; i < k; i++) {
        nextPermutation(nums);
    }

    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
        sb.append(nums[i]);
    }
    return sb.toString();
}

private void nextPermutation(int[] nums) {

}
```

1.8 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;
}
```

1.9 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] ^= '#';
}
```

1.10 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];
}
```

1.11 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>> cache) {
    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;
}
```

1.12 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 target, int index) {
    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<Integer> list) {
    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);
    }
}
```

1.13 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 target, int index) {
    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<Integer> path) {
    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);
    }
}
```

1.14 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);
    }
}
```

1.15 Combination Sum IV

Description

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

Example:

nums = `[1, 2, 3]` target = 4

The possible combination ways are:

```
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
```

Note that different sequences are counted as different combinations.

Therefore the output is 7.

Follow up:

What if negative numbers are allowed in the given array?

How does it change the problem?

What limitation we need to add to the question to allow negative numbers?

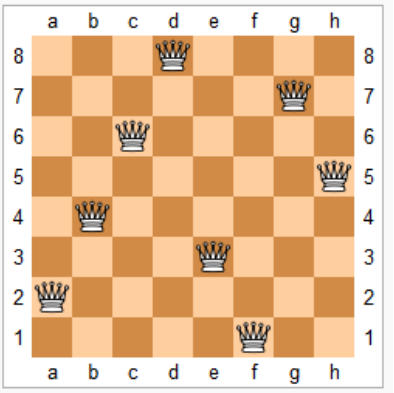
Solution

```
public int combinationSum4(int[] nums, int target) {
    int[] dp = new int[target + 1];
    // 这里排序便于之后 break
    Arrays.sort(nums);
    // 这里 0 为什么是 1 呢，我开始也不解，
    // 其实就是说如果组合中带的刚好是当前数，那么组合数只有 1 种。
    // 比如 target 为 3 的时候，遍历到了 nums 中的 3，那么只有 1 种可能。
    dp[0] = 1;
    for (int i = 1; i <= target; i++) {
        for (int num : nums) {
            if (num > i) {
                break;
            } else {
                dp[i] += dp[i - num];
            }
        }
    }
    return dp[target];
}
```

1.16 N-Queens

Description

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.



One solution to the eight queens puzzle

Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example,

There exist two distinct solutions to the 4-queens puzzle:

```
[
 [".Q..",  // Solution 1
  "...Q",
  "Q...",
  "..Q."],

 ["..Q.",  // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

Solution

```
public List<List<String>> solveNQueens(int n) {
    List<List<String>> result = new LinkedList<>();
    /**
     * 这里 f[i] 表示第 i 行皇后应该放置在第几列
     */
    int[] f = new int[n];
    dfs(f, result, 0);
    return result;
}

private void dfs(int[] f, List<List<String>> result, int row) {
    if (row == f.length) {
        List<String> list = new LinkedList<>();
        char[] c = new char[f.length];
        for (int i = 0; i < f.length; i++) {
            Arrays.fill(c, '.');
            for (int j = 0; j < f.length; j++) {
                if (j == f[i]) {
                    c[j] = 'Q';
                    break;
                }
            }
            list.add(String.valueOf(c));
        }
        result.add(list);
    }
    /**
     * 对于当前第 row 行，一列一列地尝试放置皇后，看是否合法，j 表示列，如果合法则保存到 f 中
     */
    for (int j = 0; j < f.length; j++) {
        if (isValid(f, row, j)) {
            f[row] = j;
            dfs(f, result, row + 1);
        }
    }
}

private boolean isValid(int[] f, int row, int col) {
    for (int i = 0; i < row; i++) {
        if (f[i] == col) {
            return false;
        }
        if (Math.abs(i - row) == Math.abs(f[i] - col)) {
            return false;
        }
    }
    return true;
}
```

1.17 N-Queens II

Description

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

Solution

```
private int total;

public int totalNQueens(int n) {
    int[] f = new int[n];
    dfs(f, 0);
    return total;
}

private void dfs(int[] f, int row) {
    if (row == f.length) {
        total++;
        return;
    }
    for (int j = 0; j < f.length; j++) {
        if (isValid(f, row, j)) {
            f[row] = j;
            dfs(f, row + 1);
        }
    }
}

private boolean isValid(int[] f, int row, int col) {
    for (int i = 0; i < row; i++) {
        if (f[i] == col) {
            return false;
        }
        if (Math.abs(i - row) == Math.abs(f[i] - col)) {
            return false;
        }
    }
    return true;
}
```

1.18 Subsets

Description

Given a set of distinct integers, nums, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example, If nums = `[1,2,3]`, a solution is:

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

Solution I

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new LinkedList<List<Integer>>();

    if (nums.length == 0) {
        return result;
    }

    subsets(nums, 0, result, new LinkedList<Integer>());

    return result;
}

private void subsets(int[] nums, int start, List<List<Integer>> list, List<Integer> path) {
    if (start == nums.length) {
        list.add(new LinkedList<Integer>(path));
        return;
    }

    path.add(nums[start]);
    subsets(nums, start + 1, list, path);
    path.remove(path.size() - 1);

    subsets(nums, start + 1, list, path);
}
```

Solution II

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new LinkedList<List<Integer>>();

    if (nums.length == 0) {
        return result;
    }

    Arrays.sort(nums);
    subsets(nums, 0, result, new LinkedList<Integer>());
    return result;
}

private void subsets(int[] nums, int start, List<List<Integer>> result, List<Integer> path) {
    result.add(new LinkedList<Integer>(path));
    for (int i = start; i < nums.length; i++) {
        path.add(nums[i]);
        subsets(nums, i + 1, result, path);
        path.remove(path.size() - 1);
    }
}
```

1.19 Subsets II

Description

Given a collection of integers that might contain duplicates, nums, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example, If nums = `[1,2,2]`, a solution is:

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

Solution

```
public List<List<Integer>> subsetsWithDup(int[] nums) {
    List<List<Integer>> result = new LinkedList<List<Integer>>();

    if (nums.length == 0) {
        return result;
    }
    /**
     * 千万别掉了排序
     */
    Arrays.sort(nums);
    subsetsWithDup(nums, 0, result, new LinkedList<Integer>());

    return result;
}

private void subsetsWithDup(int[] nums, int start, List<List<Integer>> list, List<Integer> path) {
    if (start == nums.length) {
        list.add(new LinkedList<Integer>(path));
        return;
    }

    path.add(nums[start]);
    subsetsWithDup(nums, start + 1, list, path);
    path.remove(path.size() - 1);

    // 既然不带当前字符，那后面如果重复的都别带，否则就是带了当前字符的子集
    for ( ; start + 1 < nums.length && nums[start + 1] == nums[start]; start++);
    subsetsWithDup(nums, start + 1, list, path);
}
```

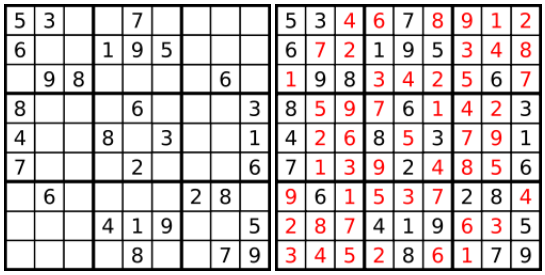
1.20 Sudoku Solver

Description

Write a program to solve a Sudoku puzzle by filling the empty cells.

Empty cells are indicated by the character `'.'`.

You may assume that there will be only one unique solution.



Solution

```
public void solveSudoku(char[][] board) {
    if (board.length == 0) {
        return;
    }
    solve(board);
}

private boolean solve(char[][] board) {
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            if (board[i][j] == '.') {
                for (char c = '1'; c <= '9'; c++) {
                    if (check(board, i, j, c)) {
                        board[i][j] = c;

                        if (solve(board)) {
                            return true;
                        } else {
                            board[i][j] = '.';
                        }
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

```
private boolean check(char[][] board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == c) {
            return false;
        }
        if (board[i][col] == c) {
            return false;
        }

        int m = (row / 3) * 3 + i / 3;
        int n = (col / 3) * 3 + i % 3;

        if (board[m][n] == c) {
            return false;
        }
    }
    return true;
}
```

1.21 Combinations

Description

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example,

If n = 4 and k = 2, a solution is:

```
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Solution

```
public List<List<Integer>> combine(int n, int k) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> path = new ArrayList<Integer>();
    combine(n, k, result, path, 1);
    return result;
}

private void combine(int n, int k, List<List<Integer>> result, List<Integer> path, int start) {
    if (k == 0) {
        result.add(new ArrayList<Integer>(path));
        return;
    }

    if (start > n) {
        return;
    }

    path.add(start);
    combine(n, k - 1, result, path, start + 1);
    path.remove(path.size() - 1);

    combine(n, k, result, path, start + 1);
}
```

1.22 Restore IP Addresses

Description

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given `"25525511135"`,

return `["255.255.11.135", "255.255.111.35"]`. (Order does not matter)

Solution

```
/**
 * 注意, 0 可以, 但是 00, 01,  010 这种是不允许的
 */
public List<String> restoreIpAddresses(String s) {
    List<String> list = new LinkedList<>();
    dfs(s, list, 0, 0, "");
    return list;
}

private void dfs(String s, List<String> list, int index, int count, String cur) {
    if (index >= s.length()) {
        if (count == 4) {
            list.add(cur);
        }
        return;
    }

    if (count == 4) {
        return;
    }

    int[][] RANGES = {
            {0, 0}, {0, 9}, {10, 99}, {100, 255}
    };
    for (int i = 1; i <= 3 && index + i <= s.length(); i++) {
        String t = s.substring(index, index + i);
        int n = Integer.parseInt(t);
        if (n >= RANGES[i][0] && n <= RANGES[i][1]) {
            dfs(s, list, index + i, count + 1, (cur.isEmpty() ? "" : cur + ".") + t);
        }
    }
}
```

1.23 Word Ladder

Description

Given two words (beginWord and endWord), and a dictionary's word list, find the length of shortest transformation sequence from beginWord to endWord, such that:

Only one letter can be changed at a time.

Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

For example,

Given:

beginWord = `"hit"`

endWord = `"cog"`

wordList = `["hot","dot","dog","lot","log","cog"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`,

return its length 5.

Note:

Return 0 if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

Solution I

```
/**
 * 要注意添加节点时要给单词从 dict 中删掉
 */
// 常规的 BFS, 耗时 141ms
public int ladderLength(String beginWord, String endWord, Set<String> wordList) {
    wordList.remove(beginWord);
    wordList.add(endWord);

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

    int ladder = 1;

    while (!queue.isEmpty()) {
        String word = queue.poll();

        StringBuilder sb = new StringBuilder(word);
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            for (int j = 0; j < 26; j++) {
                if (j + 'a' == c) {
                    continue;
                }
                sb.setCharAt(i, (char) (j + 'a'));
                String s = sb.toString();

                if (s.equals(endWord)) {
                    return ladder + 1;
                }

                if (wordList.remove(s)) {
                    next.add(s);
                }
            }
            sb.setCharAt(i, c);
        }

        if (queue.isEmpty()) {
            queue.addAll(next);
            next.clear();
            ladder++;
        }
    }

    return 0;
}
```

Solution II

```
// 采用双端 BFS, 耗时 28ms
public int ladderLength2(String beginWord, String endWord, Set<String> wordList) {
    Set<String> beginSet = new HashSet<>();
    beginSet.add(beginWord);

    Set<String> endSet = new HashSet<String>();
    endSet.add(endWord);

    int length = 1;

    while (!beginSet.isEmpty() && !endSet.isEmpty()) {
        if (beginSet.size() > endSet.size()) {
            Set<String> temp = beginSet;
            beginSet = endSet;
            endSet = temp;
        }

        Set<String> nextSet = new HashSet<String>();

        for (String word : beginSet) {
            char[] wordArr = word.toCharArray();

            for (int i = 0; i < wordArr.length; i++) {
                char c = wordArr[i];

                for (int j = 0; j < 26; j++) {
                    if ('a' + j == c) {
                        continue;
                    }
                    wordArr[i] = (char) ('a' + j);
                    String s = String.valueOf(wordArr);

                    if (endSet.contains(s)) {
                        return length + 1;
                    }

                    if (wordList.contains(s)) {
                        nextSet.add(s);
                        wordList.remove(s);
                    }
                }
                wordArr[i] = c;
            }
        }

        beginSet = nextSet;
        length++;
    }

    return 0;
}
```

1.24 Word Ladder II

Description

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:

1. Only one letter can be changed at a time
2. Each transformed word must exist in the word list. Note that beginWord is not a transformed word.

For example,

Given:

beginWord = `"hit"`

endWord = `"cog"`

wordList = `["hot","dot","dog","lot","log","cog"]`

Return

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

Note:

Return an empty list if there is no such transformation sequence.

All words have the same length.

All words contain only lowercase alphabetic characters.

You may assume no duplicates in the word list.

You may assume beginWord and endWord are non-empty and are not the same.

Analysis

这题核心就是保存前驱节点

Solution

```
/**
 * 保存前驱节点
 */
class WordNode {
    String word;
    WordNode prev;

    public WordNode(String word, WordNode pre) {
        this.word = word;
        this.prev = pre;
    }
}
```

```
public List<List<String>> findLadders(String beginWord, String endWord, Set<String> wordList) {
    List<List<String>> result = new ArrayList<List<String>>();
    LinkedList<WordNode> next = new LinkedList<>();
    HashSet<String> visited = new HashSet<String>();
    LinkedList<WordNode> queue = new LinkedList<WordNode>();
    queue.add(new WordNode(beginWord, null));

    wordList.remove(beginWord);
    wordList.add(endWord);

    while (!queue.isEmpty()) {
        WordNode top = queue.poll();
        String word = top.word;

        if (word.equals(endWord)) {
            ArrayList<String> t = new ArrayList<String>();
            for (WordNode p = top; p != null; p = p.prev) {
                t.add(0, p.word);
            }
            result.add(t);
            continue;
        }

        if (!result.isEmpty() && queue.isEmpty()) {
            break;
        }

        StringBuilder sb = new StringBuilder(word);
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            for (int j = 0; j < 26; j++) {
                if ('a' + j == c) {
                    continue;
                }
                sb.setCharAt(i, (char) ('a' + j));
                String newWord = sb.toString();
                if (wordList.contains(newWord)) {
                    next.add(new WordNode(newWord, top));
                    visited.add(newWord);
                }
            }
            sb.setCharAt(i, c);
        }

        if (queue.isEmpty()) {
            queue.addAll(next);
            next.clear();
            wordList.removeAll(visited);
        }
    }

    return result;
}
```

1.25 Palindrome Partitioning

Description

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

Return all possible palindrome partitioning of s.

For example, given s = "aab",

Return

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

Solution

```
public List<List<String>> partition(String s) {
    int n = s.length();
    List<List<String>>[] f = new LinkedList[n + 1];
    f[0] = new LinkedList<List<String>>();
    f[0].add(Collections.EMPTY_LIST);

    boolean[][] flag = new boolean[n][n];
    for (int i = 0; i < n; i++) {
        f[i + 1] = new LinkedList<List<String>>();
        for (int j = 0; j <= i; j++) {
            if (s.charAt(j) == s.charAt(i) && (j > i - 2 || flag[j + 1][i - 1])) {
                flag[j][i] = true;
                for (List<String> list : f[j]) {
                    List<String> list2 = new LinkedList<String>(list);
                    list2.add(s.substring(j, i + 1));
                    f[i + 1].add(list2);
                }
            }
        }
    }

    return f[n];
}
```

1.26 Palindrome Partitioning II

Description

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

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given s = `"aab"`,

Return 1 since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

Solution

```
/**
 * 这里 cuts[0]=-1 是为了兼容 j=i=0 的情况
 */
public int minCut(String s) {
    int n = s.length();
    int[] cuts = new int[n + 1];
    cuts[0] = -1;
    boolean[][] f = new boolean[n][n];
    for (int i = 0; i < n; i++) {
        int cut = Integer.MAX_VALUE;
        for (int j = 0; j <= i; j++) {
            if (s.charAt(j) == s.charAt(i) && (j > i - 2 || f[j + 1][i - 1])) {
                f[j][i] = true;
                cut = Math.min(cut, cuts[j] + 1);
            }
        }
        cuts[i + 1] = cut;
    }
    return cuts[n];
}
```

1.27 Palindrome Permutation

Description

Given a string, determine if a permutation of the string could form a palindrome.

For example, `"code" -> False, "aab" -> True, "carerac" -> True`.

Solution

```
public boolean canPermutePalindrome(String s) {
    int len = s.length();
    int[] count = new int[256];
    for (char c : s.toCharArray()) {
        count[c]++;
    }
    int num = 0;
    for (int i = 0; i < count.length; i++) {
        num += (count[i] >> 1) << 1;
    }
    if (num < len) {
        num++;
    }
    return num == len;
}
```

1.28 Palindrome Permutation II

Description

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:

Given s = `"aabb"`, return `["abba", "baab"]`.

Given s = `"abc"`, return [].

Solution

```
public static List<String> generatePalindromes(String s) {
    int[] counts = new int[256];
    for (char c : s.toCharArray()) {
        counts[c]++;
    }
    List<String> list = new LinkedList<>();
    StringBuilder sb = new StringBuilder();
    char single = 0;
    for (int i = 0; i < counts.length; i++) {
        if (counts[i] % 2 != 0) {
            if (single != 0) {
                return list;
            }
            single = (char) i;
            counts[i] = ((counts[i] >> 1) << 1);
        }
        for (int j = 0; j < counts[i]; j += 2) {
            sb.append((char)i);
        }
    }

    helper(sb, "" + (single != 0 ? single : ""), list);
    return list;
}

private static void helper(StringBuilder sb, String cur, List<String> list) {
    if (sb.length() == 0) {
        list.add(cur);
        return;
    }

    for (int i = 0; i < sb.length(); i++) {
        if (i > 0 && sb.charAt(i) == sb.charAt(i - 1)) {
            continue;
        }
        char c = sb.charAt(i);
        sb.deleteCharAt(i);
        helper(sb, c + cur + c, list);
        sb.insert(i, c);
    }
}
```

1.29 Count Numbers with Unique Digits

Description

Given a non-negative integer n, count all numbers with unique digits, `x`, where 0 <= x < 10n.

Example:

Given n = 2, return 91. (The answer should be the total numbers in the range of 0 <= x < 100, excluding `[11,22,33,44,55,66,77,88,99]`)

Solution

```
public int countNumbersWithUniqueDigits(int n) {
    int[] result = new int[n + 1];
    result[0] = 1;
    for (int i = 1, t = 9; i <= n; i++) {
        result[i] = result[i - 1] + t;
        t *= 10 - i;
    }
    return result[n];
}
```

1.30 Generalized Abbreviation

Description

Write a function to generate the generalized abbreviations of a word.

Example:

Given word = ”word”, return the following list (order does not matter):

`["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2", "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]`

Solution

```
public List<String> generateAbbreviations(String word) {
    List<String> list = new LinkedList<>();
    helper(word, 0, list, "", 0);
    return list;
}

private void helper(String word, int idx, List<String> list, String cur, int count) {
    if (idx == word.length()) {
        if (count > 0) {
            cur += count;
        }
        list.add(cur);
        return;
    }

    helper(word, idx + 1, list, cur, count + 1);
    helper(word, idx + 1, list, cur + (count > 0 ? count : "") + word.charAt(idx), 0);
}
```

1.31 Binary Watch

Description

A binary watch has 4 LEDs on the top which represent the hours (0-11), and the 6 LEDs on the bottom represent the minutes (0-59).

Each LED represents a zero or one, with the least significant bit on the right.

For example, the above binary watch reads `"3:25"`.

Given a non-negative integer n which represents the number of LEDs that are currently on, return all possible times the watch could represent.

Example:

Input: n = 1

Return: `["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]`

Note:

The order of output does not matter.

The hour must not contain a leading zero, for example `"01:00"` is not valid, it should be `"1:00"`.

The minute must be consist of two digits and may contain a leading zero, for example `"10:2"` is not valid, it should be `"10:02"`.

Solution

```
public List<String> readBinaryWatch(int num) {
    int[] f = new int[] {
            1, 2, 4, 8, 1, 2, 4, 8, 16, 32
    };

    List<String> list = new ArrayList<String>();
    readBinaryWatch(num, f, 0, 0, 0, list);

    return list;
}
```

```
public void readBinaryWatch(int num, int[] f, int index, int hour, int minute, List<String> list) {
    if (num == 0 && hour < 12 && minute < 60) {
        list.add(String.format("%d:%02d", hour, minute));
        return;
    }

    if (index >= f.length) {
        return;
    }

    int nextHour = index <= 3 ? hour + f[index] : hour;
    int nextMinute = index > 3 ? minute + f[index] : minute;
    readBinaryWatch(num - 1, f, index + 1, nextHour, nextMinute, list);
    readBinaryWatch(num, f, index + 1, hour, minute, list);
}
```

1.32 Add and Search Word

Description

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

For example:

```
addWord("bad")
addWord("dad")
addWord("mad")
search("pad") -> false
search("bad") -> true
search(".ad") -> true
search("b..") -> true
```

Note:

You may assume that all words are consist of lowercase letters a-z.

Solution

```
TriNode root;

class TriNode {
    TriNode[] nodes = new TriNode[26];
    String word;
}

public AddAndSearchWord() {
    root = new TriNode();
}

public void addWord(String word) {
    TriNode node = root;
    for (char c : word.toCharArray()) {
        if (node.nodes[c - 'a'] == null) {
            node.nodes[c - 'a'] = new TriNode();
        }
        node = node.nodes[c - 'a'];
    }
    node.word = word;
}

public boolean search(String word) {
    return search(root, word);
}
```

```
private boolean search(TriNode node, String word) {
    if (node == null) {
        return false;
    }

    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);

        if (c != '.') {
            if (node.nodes[c - 'a'] == null) {
                return false;
            }
            node = node.nodes[c - 'a'];
        } else {
            String s = word.substring(i + 1);
            for (char cc = 'a'; cc <= 'z'; cc++) {
                if (search(node.nodes[cc - 'a'], s)) {
                    return true;
                }
            }
            return false;
        }
    }

    return node.word != null;
}
```

1.33 Factor Combinations

Descriptor

Numbers can be regarded as product of its factors. For example,

```
8 = 2 x 2 x 2;
  = 2 x 4.
```

Write a function that takes an integer n and return all possible combinations of its factors.

Note:

You may assume that n is always positive.

Factors should be greater than 1 and less than n.

Examples:

input: 1

output: `[]`

input: 37

output: `[]`

input: 12

output:

```
[
  [2, 6],
  [2, 2, 3],
  [3, 4]
]
```

input: 32

output:

```
[
  [2, 16],
  [2, 2, 8],
  [2, 2, 2, 4],
  [2, 2, 2, 2, 2],
  [2, 4, 4],
  [4, 8]
]
```

Solution

```
public List<List<Integer>> getFactors(int n) {
    List<List<Integer>> result = new LinkedList<>();
    dfs(n, 2, result, new LinkedList<>());
    return result;
}

private void dfs(int n, int cur, List<List<Integer>> result, List<Integer> list) {
    if (n <= 1) {
        if (list.size() > 1) {
            result.add(new LinkedList<>(list));
        }
        return;
    }

    for (int i = cur; i <= n; i++) {
        if (n % i == 0) {
            list.add(i);
            dfs(n / i, i, result, list);
            list.remove(list.size() - 1);
        }
    }
}
```

1.34 Valid Word Square

Description

Given a sequence of words, check whether it forms a valid word square.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 <= k < max(numRows, numColumns).

Note:

1. The number of words given is at least 1 and does not exceed 500.
2. Word length will be at least 1 and does not exceed 500.
3. Each word contains only lowercase English alphabet a-z.

Example:

Input:

```
[
  "abcd",
  "bnrt",
  "crmy",
  "dtye"
]
```

Output:

true

Explanation:

The first row and first column both read ”abcd”.

The second row and second column both read ”bnrt”.

The third row and third column both read ”crmy”.

The fourth row and fourth column both read ”dtye”.

Therefore, it is a valid word square.

Solution

```
public boolean validWordSquare(List<String> words) {
    for (int i = 0; i < words.size(); i++) {
        String word = words.get(i);
        if (word.length() > words.size()) {
            return false;
        }
        for (int j = 0; j < word.length(); j++) {
            String s = words.get(j);
            if (i >= s.length() || word.charAt(j) != s.charAt(i)) {
                return false;
            }
        }
    }
    return true;
}
```

1.35 Word Squares

Description

Given a set of words (without duplicates), find all word squares you can build from them.

A sequence of words forms a valid word square if the kth row and column read the exact same string, where 0 <= k < max(numRows, numColumns).

For example, the word sequence `["ball","area","lead","lady"]` forms a word square because each word reads the same both horizontally and vertically.

```
b a l l
a r e a
l e a d
l a d y
```

Note:

1. There are at least 1 and at most 1000 words.
2. All words will have the exact same length.
3. Word length is at least 1 and at most 5.
4. Each word contains only lowercase English alphabet a-z.

Example:

Input:

```
["area","lead","wall","lady","ball"]
```

Output:

```
[
  [ "wall",
    "area",
    "lead",
    "lady"
  ],
  [ "ball",
    "area",
    "lead",
    "lady"
  ]
]
```

Explanation:

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

Solution

```
public List<List<String>> wordSquares(String[] words) {
    List<List<String>> ret = new ArrayList<List<String>>();
    if (words.length == 0 || words[0].length() == 0) {
        return ret;
    }
    Map<String, Set<String>> map = new HashMap<>();
    int squareLen = words[0].length();
    for (int i = 0; i < words.length; i++) {
        /**
         * 注意这里空字符也算前缀, 所以 j 从-1 开始
         */
        for (int j = -1; j < words[0].length(); j++) {
            String prefix = words[i].substring(0, j + 1);
            Set<String> set = map.get(prefix);
            if (set == null) {
                set = new HashSet<String>();
                map.put(prefix, set);
            }
            set.add(words[i]);
        }
    }
    helper(ret, new ArrayList<String>(), squareLen, map);
    return ret;
}

public void helper(List<List<String>> ret, List<String> cur, int total, Map<String, Set<String>> map) {
    if (cur.size() == total) {
        ret.add(new ArrayList<String>(cur));
        return;
    }
    // build search string
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < cur.size(); i++) {
        /**
         * 下一个单词的 prefix 必然是当前这些单词的第 cur.size 列
         */
        sb.append(cur.get(i).charAt(cur.size()));
    }
    // bachtracking
    Set<String> cand = map.get(sb.toString());
    if (cand == null) {
        return;
    }
    for (String str : cand) {
        cur.add(str);
        helper(ret, cur, total, map);
        cur.remove(cur.size() - 1);
    }
}
```

1.36 Beautiful Arrangement

Description

Suppose you have N integers from 1 to N. We define a beautiful arrangement as an array that is constructed by these N numbers successfully if one of the following is true for the ith position (1 <= i <= N) in this array:

1. The number at the ith position is divisible by i.
2. i is divisible by the number at the ith position.

Now given N, how many beautiful arrangements can you construct?

Example:

Input: 2

Output: 2

Explanation:

The first beautiful arrangement is [1, 2]:

Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).

Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).

The second beautiful arrangement is [2, 1]:

Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).

Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.

Note:

N is a positive integer and will not exceed 15.

Solution

```
public int countArrangement(int N) {
    int[] count = new int[1];
    helper(N, 1, new boolean[N + 1], count);
    return count[0];
}

private void helper(int N, int k, boolean[] visited, int[] count) {
    if (k > N) {
        count[0]++;
        return;
    }

    for (int i = 1; i <= N; i++) {
        if (!visited[i] && (i % k == 0 || k % i == 0)) {
            visited[i] = true;
            helper(N, k + 1, visited, count);
            visited[i] = false;
        }
    }
}
```

1.37 Flip Game

Description

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "--". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

For example, given s = "++++", after one move, it may become one of the following states:

```
[
  "--++",
  "+--+",
  "++--"
]
```

If there is no valid move, return an empty list [].

Solution

```
public List<String> generatePossibleNextMoves(String s) {
    List<String> result = new LinkedList<>();
    for (int i = 0; i < s.length(); ) {
        int index = s.indexOf("++", i);
        if (index >= i) {
            result.add(s.substring(0, index) + "--" + s.substring(index + 2));
            i = index + 1;
        } else {
            break;
        }
    }
    return result;
}
```

1.38 Flip Game II

Description

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive `"++"` into `"--"`. The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

For example, given s = `"++++"`, return true. The starting player can guarantee a win by flipping the middle `"++"` to become `"+--+"`.

Follow up:

Derive your algorithm's runtime complexity.

Solution

```
public boolean canWin(String s) {
    for (int i = 0; i < s.length(); ) {
        int index = s.indexOf("++", i);
        if (index >= i) {
            String t = s.substring(0, index) + "--" + s.substring(index + 2);
            if (!canWin(t)) {
                return true;
            }
            i = index + 1;
        } else {
            break;
        }
    }
    return false;
}
```

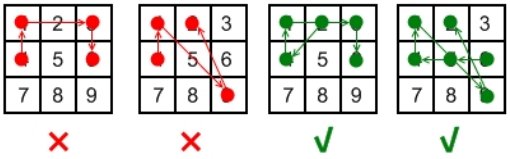
1.39 Android Unlock Patterns

Description

Given an Android 3x3 key lock screen and two integers m and n, where $1 \leq m \leq n \leq 9$, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

Rules for a valid pattern:

1. Each pattern must connect at least m keys and at most n keys.
2. All the keys must be distinct.
3. If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
4. The order of keys used matters.



Explanation:

```
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
```

Invalid move: `4 - 1 - 3 - 6`

Line `1 - 3` passes through key 2 which had not been selected in the pattern.

Invalid move: `4 - 1 - 9 - 2`

Line `1 - 9` passes through key 5 which had not been selected in the pattern.

Valid move: `2 - 4 - 1 - 3 - 6`

Line `1 - 3` is valid because it passes through key 2, which had been selected in the pattern

Valid move: `6 - 5 - 4 - 1 - 9 - 2`

Line `1 - 9` is valid because it passes through key 5, which had been selected in the pattern.

Example:

Given m = 1, n = 1, return 9.

Solution

```
int DFS(boolean vis[], int[][] skip, int cur, int remain) {
    if(remain < 0) return 0;
    if(remain == 0) return 1;
    vis[cur] = true;
    int rst = 0;
    for(int i = 1; i <= 9; ++i) {
        // If vis[i] is not visited and (two numbers are adjacent or skip number is already visited)
        if(!vis[i] && (skip[cur][i] == 0 || (vis[skip[cur][i]]))) {
            rst += DFS(vis, skip, i, remain - 1);
        }
    }
    vis[cur] = false;
    return rst;
}

public int numberOfPatterns(int m, int n) {
    // Skip array represents number to skip between two pairs
    int skip[][] = new int[10][10];
    skip[1][3] = skip[3][1] = 2;
    skip[1][7] = skip[7][1] = 4;
    skip[3][9] = skip[9][3] = 6;
    skip[7][9] = skip[9][7] = 8;
    skip[1][9] = skip[9][1] = skip[2][8] = skip[8][2] = skip[3][7] = skip[7][3] = skip[4][6] = skip[6][4] = 5;
    boolean vis[] = new boolean[10];
    int rst = 0;
    // DFS search each length from m to n
    for(int i = m; i <= n; ++i) {
        rst += DFS(vis, skip, 1, i - 1) * 4;    // 1, 3, 7, 9 are symmetric
        rst += DFS(vis, skip, 2, i - 1) * 4;    // 2, 4, 6, 8 are symmetric
        rst += DFS(vis, skip, 5, i - 1);        // 5
    }
    return rst;
}
```