Chapter 6 Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably **constraint satisfaction problems**, that **incrementally builds** candidates to the solutions, and **abandons** a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution. -- wiki

computational problems 没有合适的优化解法，需要考虑到所有的可能性

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

Example:

```
Input: "25525511135"
Output: ["255.255.11.135", "255.255.111.35"]
```

最直接的方法（多重循环）

```
public List<String> restoreIpAddresses(String s) {
    List<String> res = new ArrayList<String>();
    int len = s.length();
    for(int i = 1; i<4 && i<len-2; i++){
        for(int j = i+1; j<i+4 && j<len-1; j++){
            for(int k = j+1; k<j+4 && k<len; k++){
                String s1 = s.substring(0,i), s2 = s.substring(i,j), s3 =
s.substring(j,k), s4 = s.substring(k,len);
                if(isValid(s1) && isValid(s2) && isValid(s3) && isValid(s4)){
                    res.add(s1+"."+s2+"."+s3+"."+s4);
                }
            }
        }
    }
    return res;
}

public boolean isValid(String s){
    if(s.length()>3 || s.length()==0 || (s.charAt(0)=='0' && s.length()>1) ||
Integer.parseInt(s)>255)
        return false;
    return true;
}
```

递归解法（Depth-First-Search -》以DFS的方式搜索/尝试所有可能性）

```
public List<String> restoreIpAddresses(String s) {
```

```
    List<String> solutions = new ArrayList<String>();
    restoreIp(s, solutions, 0, "", 0);
    return solutions;
}

private void restoreIp(String ip, List<String> solutions, int idx, String restored,
int count) {
    if (count > 4) return;
    if (count == 4 && idx == ip.length()) solutions.add(restored);

    for (int i=1; i<4; i++) {
        if (idx+i > ip.length()) break; // "255.255.111.3" idx = 9, i = 1, 2, 3

        String s = ip.substring(idx,idx+i);

        if ((s.startsWith("0") && s.length()>1) || (i == 3 && Integer.parseInt(s)
>= 256)) continue;

        restoreIp(ip, solutions, idx+i, restored+s+(count==3?"" : "."), count + 1);
    }
}
```

https://leetcode.com/problems/restore-ip-addresses/discuss/168080/concise-Java-backtracking-solution

Generate Parentheses

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:

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

Depth-First-Search解法

```
public List<String> generateParenthesis(int n) {
    List<String> ans = new ArrayList();
    backtrack(ans, "", 0, 0, n);
    return ans;
}

public void backtrack(List<String> ans, String cur, int open, int close, int max){
    if (str.length() == max * 2) {
        ans.add(cur);
```

```
        return;
    }

    if (open < max)
        backtrack(ans, cur+"(", open+1, close, max);
    if (close < open)
        backtrack(ans, cur+")", open, close+1, max);
}
```

[Word Search II]

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.

Example:

```
Input:
words = ["oath","pea","eat","rain"] and board =
[
  ['o','a','a','n'],
  ['e','t','a','e'],
  ['i','h','k','r'],
  ['i','f','l','v']
]

Output: ["eat","oath"]
```

DFS + Trie 解法：

- 穷举所有可能性 -》DFS
- 判断每一种可能性是否Valid -》Trie
- Time Complexity: O(n*m*n*m) = 矩阵字符数 * 每一次DFS时间

如何对于board[0][0]元素进行DFS：

o -> 上面没有元素
  -> 左边没有元素
  -> 右边为a -> 上面没有元素
  -> 左边的已遍历
  -> 右边为a, 但是Tire中没有 oaa 路径
  -> 下面为t（In Trie） -> 上面的已遍历
  -> 左边为e，但是Tire中没有 oate 路径
  -> 右边为a，但是Tire中没有 oata 路径
  -> **下面为h, 在Trie中找到了该节点 oath**, 现在返回
  因为Tire中没有后续节点
  -> 下面为e，但是Tire中没有 oe 路径

```
public List<String> findWords(char[][] board, String[] words) {
```

```
    List<String> res = new ArrayList<>();
    TrieNode root = buildTrie(words);
    for (int i = 0; i < board.length; i++) {
        for (int j = 0; j < board[0].length; j++) {
            dfs (board, i, j, root, res);
        }
    }
    return res;
}

# TrieNode p指代当前已经遍历到的可行路径
public void dfs(char[][] board, int i, int j, TrieNode p, List<String> res) {
    char c = board[i][j];
    if (c == '#' || p.next[c - 'a'] == null) return;
    p = p.next[c - 'a'];
    if (p.word != null) {   // found one
        res.add(p.word);
        p.word = null;     // de-duplicate
    }

    board[i][j] = '#';
    if (i > 0) dfs(board, i - 1, j ,p, res);
    if (j > 0) dfs(board, i, j - 1, p, res);
    if (i < board.length - 1) dfs(board, i + 1, j, p, res);
    if (j < board[0].length - 1) dfs(board, i, j + 1, p, res);
    board[i][j] = c;
}

public TrieNode buildTrie(String[] words) {
    TrieNode root = new TrieNode();
    for (String w : words) {
        TrieNode p = root;
        for (char c : w.toCharArray()) {
            int i = c - 'a';
            if (p.next[i] == null) p.next[i] = new TrieNode();
            p = p.next[i];
       }
       p.word = w;
    }
    return root;
}

class TrieNode {
    TrieNode[] next = new TrieNode[26];
    String word;
}
```

Combination问题

[Combination Sum]
Given a set of candidate numbers (candidates) (**without duplicates**) and a target number (target), find all unique combinations in candidates where the candidate numbers sums to target.

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

Note:
All numbers (including target) will be positive integers.
The solution set must not contain duplicate combinations.

Example:

```
Input: candidates = [2,3,6,7], target = 7,
A solution set is:
[
  [7],
  [2,2,3]
]

Input: candidates = [2,3,5], target = 8,
A solution set is:
[
  [2,2,2,2],
  [2,3,3],
  [3,5]
]
```

```
public List<List<Integer>> combinationSum(int[] nums, int target) {
    List<List<Integer>> list = new ArrayList<>();
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int []
nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) {
        list.add(new ArrayList<>(tempList));
    } else {
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain - nums[i], i);
            tempList.remove(tempList.size() - 1);
        }
```

```
    }
}
```

Similar Questions:
Letter Combinations of a Phone Number
Combination Sum II
Combinations
Combination Sum III

Permutation问题

Permutations
Given a collection of distinct integers, return all possible permutations.

Example:

```
Input: [1,2,3]
Output:
[
  [1,2,3],
  [1,3,2],
  [2,1,3],
  [2,3,1],
  [3,1,2],
  [3,2,1]
]
```

Time Complexity: O(n!)

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

private void backtrack(List<List<Integer>> list, List<Integer> tempList, int []
nums){
   if(tempList.size() == nums.length){
      list.add(new ArrayList<>(tempList));
   } else{
      for(int i = 0; i < nums.length; i++){
         if(tempList.contains(nums[i])) continue; // element already exists, skip
         tempList.add(nums[i]);
         backtrack(list, tempList, nums);
         tempList.remove(tempList.size() - 1);
      }
```

```
    }
}
```

Similar Questions:
Permutations II
Beautiful Arrangement

Subset问题

Subsets
Given a set of **distinct integers**, nums, return all possible subsets (the **power set**).
Note: The solution set must not contain duplicate subsets.

Example:

```
Input: nums = [1,2,3]
Output:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

```
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, 0);
    return list;
}

private void backtrack(List<List<Integer>> list , List<Integer> tempList, int []
nums, int start) {
    list.add(new ArrayList<>(tempList));
    for(int i = start; i < nums.length; i++){
        tempList.add(nums[i]);
        backtrack(list, tempList, nums, i + 1);
        tempList.remove(tempList.size() - 1);
    }
}
```

Similar Questions:

Subsets II
Increasing Subsequences

现实中，也就是实践中，我们是否会这样去设计或者写Permutation，Combination和Subset呢？Google Guava 是 Google开源出来的一个 Java library，专门用于提供很多辅助的功能，让你快速的去写逻辑代码同时提供high performance。

Google Guava Permutation:
https://github.com/google/guava/blob/master/guava/src/com/google/common/collect/Collections2.java#L622

DFS and Memoization

Target Sum
You are given a list of non-negative integers, a1, a2, ..., an, and a target, S. Now you have 2 symbols + and -. For each integer, you should choose one from + and - as its new symbol.

Find out how many ways to assign symbols to make sum of integers equal to target S.

其中，所有值相加只和不会超过1000

Example:

```
Input: nums is [1, 1, 1, 1, 1], S is 3.
Output: 5
Explanation:

-1+1+1+1+1 = 3
+1-1+1+1+1 = 3
+1+1-1+1+1 = 3
+1+1+1-1+1 = 3
+1+1+1+1-1 = 3

There are 5 ways to assign symbols to make the sum of nums be target 3.
```

Brute Force

- 可以穷举出所有可能性
- time complexity: O(2^n)

```
int count = 0;

public int findTargetSumWays(int[] nums, int S) {
    calculate(nums, 0, 0, S);
    return count;
}

public void calculate(int[] nums, int i, int curSum, int S) {
    if (i == nums.length) {
```

```
        if (curSum == S)
            count++;
    } else {
        calculate(nums, i + 1, curSum + nums[i], S);
        calculate(nums, i + 1, curSum - nums[i], S);
    }
}
```

用 memoization 来优化

- 使用Map/Array来存储子问题计算结果，从而避免子问题的重复计算

```
public int findTargetSumWays(int[] nums, int S) {
    // memo[i][0] 前i个数能组成-1000的可能性；memo[i][1000] 前i个数能组成0的可能性; memo[0][2001] 前i个数能组成1000的可能性
    int[][] memo = new int[nums.length][2001];
    for (int[] row: memo)
        Arrays.fill(row, Integer.MIN_VALUE);

    return calculate(nums, 0, 0, S, memo);
}

public int calculate(int[] nums, int i, int curSum, int S, int[][] memo) {
    if (i == nums.length) {
        if (curSum == S)
            return 1;
        else
            return 0;
    } else {
        // 避免重复计算子问题，即重复计算前i个数能组成的curSum情况的个数
        if (memo[i][curSum + 1000] != Integer.MIN_VALUE) {
            return memo[i][curSum + 1000];
        }
        int subtract = calculate(nums, i + 1, curSum + nums[i], S, memo);
        int add = calculate(nums, i + 1, curSum - nums[i], S, memo);
        memo[i][curSum + 1000] = add + subtract;
        return memo[i][curSum + 1000];
    }
}
```

其它问题：

Partition to K Equal Sum Subsets

Can I Win