回溯-排列组合篇

在我看来，回溯的题目大体可以分为二种。第一种为排列组合类回溯题目，较为简单，根据模板稍作变化即可。第二种包含 N-Queens，字符串的操作和处理题目，较为复杂，或者需要额外的检查方法，或者判断条件多变，但也万变不离模板。下面提供一个黄金模板，[所有]回溯题目均可套用。

在 LeetCode 中关于排列组合的题目有：

1. Subsets
2. Subsets II
3. Permutations
4. Permutations II
5. Combinations
6. Combination Sum
7. Combination Sum II
8. Letter Combinations of A Phone Number

回溯题目[黄金]模板：

```
public ArrayList<ArrayList<Integer>> combinationSum2(int[] num, int target) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length == 0){
        return res;
    }
    Arrays.sort(num); // ①
    ArrayList<Integer> item = new ArrayList<Integer>(); // ②
    helper(num, 0, target, item, res); // ③
    return res;
}

private void helper(int[] num, int start, int target, ArrayList<Integer> item, ArrayList<ArrayList<Integer>> res){
    if(target < 0){
        return;
    }
```

```
        if(target == 0){ // ④
            res.add(new ArrayList<Integer>(item));
            return;
        }
        for(int i = start; i < num.length; i++){
            if(i > start && num[i] == num[i - 1]){ // ⑤
                continue;
            }
            item.add(num[i]);
            helper(num, i + 1, target - num[i], item, res);
            item.remove(item.size() - 1);
        }
    }
```

需要思考和更改的部分：

① 集合对顺序是否有要求。当题目要求结果有序或要去掉重复值时，往往需要对集合排序。

② 定义一个合适的存储临时结果的容器，可能为数组，List，StringBuilder等。

③ 这里要考虑每次递归传入的参数是什么。往往参数的构成是：

原始参数 + 临时容器item + 最终结果res + 标记数组used + 字典dict + [start指针]

④ 这里需要根据题意写递归出口。做出合理的判断和对应的输出。

⑤ 此处是题目变形的关键，往往用来去重复，查看是否已用，查看是否合理等。

Warning：所有回溯题目一律使用模板来解，其他方法只为备用方案。

1. Subsets，求子集问题是经典的NP问题，复杂度上我们就无法强求了，肯定是非多项式量级的。一般来说这个问题有两种解法：递归和迭代。

递归解法直接使用模板即可。

迭代解法主要需要理解，在原有集合加入一个新的数字，如何得到包含新数字的所有子集。其实就是在原有的集合中对每集合中的每个元素都加入新元素得到子集，然后放入原有集合中（原来的集合中的元素不用删除，因为他们也是合法子集）。时间和空间都是取决于结果的数量，也就是O(2^n)。

位运算解法。对于这道题，因为集合中没有重复的数字，所以不用考虑结果重复的问题，可以使用位运算这一巧妙的方法来解，思路清晰，代码简洁，但不是通法。

```
// Solution 1 – Classic Model
public ArrayList<ArrayList<Integer>> subsets(int[] S) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(S == null || S.length == 0){
        return res;
    }
    Arrays.sort(S);
    ArrayList<Integer> item = new ArrayList<Integer>();
    helper(S, 0, item, res);
    return res;
}

private void helper(int[] S, int start, ArrayList<Integer> item,
ArrayList<ArrayList<Integer>> res){
    res.add(new ArrayList<Integer>(item));
    for(int i = start; i < S.length; i++){
        item.add(S[i]);
        helper(S, i + 1, item, res);
        item.remove(item.size() - 1);
    }
}
// Solution 2 – Iteration
public ArrayList<ArrayList<Integer>> subsets(int[] S) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(S == null || S.length == 0){
        return res;
    }
    Arrays.sort(S);
    ArrayList<Integer> item = new ArrayList<Integer>();
    res.add(item);
    for(int i = 0; i < S.length; i++){
        int size = res.size();
        for(int j = 0; j < size; j++){
            item = new ArrayList<Integer>(res.get(j));
            item.add(S[i]);
            res.add(item);
        }
    }
    return res;
}
```

```
    // Solution 3 – Bit Manipulation
    public ArrayList<ArrayList<Integer>> subsets(int[] S) {
        if(S == null || S.length == 0){
            return null;
        }
        Arrays.sort(S);
        int total = 1 << S.length;
        ArrayList<ArrayList<Integer>> res = new
ArrayList<ArrayList<Integer>>(total);
        for(int i = 0; i < total; i++){
            ArrayList<Integer> item = new ArrayList<Integer>();
            for(int j = 0; j < S.length; j++){
                if(((i >> j) & 1) == 1){
                    item.add(S[j]);
                }
            }
            res.add(item);
        }
        return res;
    }
```

2. Subsets II，这个问题同样有两种解法：递归和迭代。

递归解法，主要注意跟上一道题的区别，就是对重复结果的排除。我们只关心取多少个x元素，不关心取哪几个。所以规定当前循环必须从第一个x开始连续取，不可以跳过第一个x直接取第二个x，这样会导致重复结果。

迭代解法，为了避免重复，每当遇到重复元素的时候我们就只把当前结果集的后半部分加上当前元素加入到结果集中，因为后半部分就是上一步中加入这个元素的所有子集，上一步这个元素已经加入过了，前半部分如果再加就会出现重复。

```
    // Solution 1 – Classic Model
    public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        if(num == null || num.length == 0){
            return res;
        }
        Arrays.sort(num);
        ArrayList<Integer> item = new ArrayList<Integer>();
        helper(num, 0, item, res);
```

```
        return res;
    }

    private void helper(int[] num, int start, ArrayList<Integer> item,
ArrayList<ArrayList<Integer>> res){
        res.add(new ArrayList<Integer>(item));
        for(int i = start; i < num.length; i++){
            if(i > 0 && num[i] == num[i - 1] && i != start){
                continue;
            }
            item.add(num[i]);
            helper(num, i + 1, item, res);
            item.remove(item.size() - 1);
        }
    }
    // Solution 2 - Iteration
    public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        if(num == null || num.length == 0){
            return res;
        }
        Arrays.sort(num);
        ArrayList<Integer> item = new ArrayList<Integer>();
        res.add(item);
        int start = 0;
        for(int i = 0; i < num.length; i++){
            int size = res.size();
            for(int j = start; j < size; j++){
                item = new ArrayList<Integer>(res.get(j));
                item.add(num[i]);
                res.add(item);
            }
            if(i < num.length - 1 && num[i] == num[i + 1]){
                start = size;
            } else {
                start = 0;
            }
        }
        return res;
    }
```

3. Permutations，方法还是套用模板，使用一个循环递归处理子问题。区别是这里并不是一直往后推进的，前面的数有可能放到后面，所以我们需要维护一个used数组来表示该元素是否已经在当前结果中，时间复杂度还是NP问题的指数量级。

注意在实现中有一个细节，就是在递归函数的前面，我们分别设置了used[i]的标记，表明改元素被使用，并且把元素加入到当前结果中，而在递归函数之后，我们把该元素从结果中移除，并把标记置为false，这个我们可以称为“保护现场”，递归函数必须保证在进入和离开函数的时候，变量的状态是一样的，这样才能保证正确性。这种方法在很多题目（几乎所有NP问题）中一直用到。

代码如下：

```
// Solution 1 – Classic Model
public ArrayList<ArrayList<Integer>> permute(int[] num) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length == 0){
        return res;
    }
    ArrayList<Integer> item = new ArrayList<Integer>();
    boolean[] used = new boolean[num.length];
    helper(num, used, item, res);
    return res;
}

private void helper(int[] num, boolean[] used, ArrayList<Integer> item, ArrayList<ArrayList<Integer>> res){
    if(item.size() == num.length){
        res.add(new ArrayList<Integer>(item));
        return;
    }
    for(int i = 0; i < num.length; i++){
        if(!used[i]){
            used[i] = true;
            item.add(num[i]);
            helper(num, used, item, res);
            item.remove(item.size() - 1);
            used[i] = false;
        }
    }
}
```

迭代一般就是要清楚每次加入一个新的元素，我们应该做什么，这里其实还是比较清晰的，假设有了前i个元素的所有permutation，当第i + 1个元素加进来时，我们要做的就是将这个元素带入每一个之前的结果，并且放到每一个结果的各个位置中。因为之前的结果没有重复，所以带入新元素之后也不会有重复（当然这里假设数字集合本身是没有重复的元素的）。上面的代码有时候乍一看可能觉得只有两层循环，但因为注意第二层循环是对于res进行遍历的，而res是一直在以平方量级增长的，所以总的时间复杂度还是指数量级以上的。

```
// Solution 2 - Iteration
public ArrayList<ArrayList<Integer>> permute(int[] num) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length == 0){
        return res;
    }
    ArrayList<Integer> item = new ArrayList<Integer>();
    item.add(num[0]);
    res.add(item);
    for(int i = 1; i < num.length; i++){
        ArrayList<ArrayList<Integer>> curRes = new
ArrayList<ArrayList<Integer>>();
        for(int j = 0; j < res.size(); j++){
            ArrayList<Integer> cur = res.get(j);
            for(int k = 0; k <= cur.size(); k++){
                cur.add(k, num[i]);
                curRes.add(new ArrayList<Integer>(cur));
                cur.remove(k);
            }
        }
        res = curRes;
    }
    return res;
}
```

做题时的感悟：

Solution 1 - BackTracking

1. 这应该是最简单的循环递归问题了。我们可以直接用 item.size()来衡量有多少元素加入到结果中了。如果 item.size() == num.length，储存结果即可。

2. 在把 item 加入 res 中的时候，切记要新建一个 ArrayList，代码如下：

```
if(item.size() == num.length){
    res.add(new ArrayList<Integer>(item));
    return;
}
```

否则在回溯的时候会改变 res 中 item 的值，就产生了错误的结果。

3. 回溯时，要有“保护现场”的思想。递归函数必须保证在进入和离开函数的时候，变量的状态是一样的。

Solution 2 - Iteration

1. 保持清晰的思路是递归解题的关键。每个循环有每个循环的作用。第一层，插入 num 中每一个元素；第二层，从 res 中逐个取出已有的 ArrayList；第三层，将 num[i]插入到第二层取出的 ArrayList 的每一个空位中。

2. 其中第三层循环时，有 2 种写法。分别如下：

```
for(int k = 0; k <= cur.size(); k++){
    list = new ArrayList<Integer>(cur);
    list.add(k, num[i]);
    curRes.add(list);
}
```

```
for(int k = 0; k <= cur.size(); k++){
    cur.add(k, num[i]);
    curRes.add(new ArrayList<Integer>(cur));
    cur.remove(k);
}
```

哪种更好有待考究。经测试，所用时间基本一致，但第二种会省空间吗？好像不会。

4. Permutations II，这个题跟Permutations非常类似，唯一的区别就是在这个题目中元素集合可以出现重复。那么如何避免这种重复呢？方法就是对于重复的元素循环时跳过递归函数的调用，只对第一个未被使用的进行递归，那么这一次结果会出现在第一个的递归函数结果中，而后面重复的会被略过。如果第一个重复元素前面的元素还没在当前结果中，那么我们不需要进行递归。

```
// Solution 1 - Classic Model
public ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length == 0){
        return res;
    }
    Arrays.sort(num);
    ArrayList<Integer> item = new ArrayList<Integer>();
    boolean[] used = new boolean[num.length];
    helper(num, used, item, res);
    return res;
}

private void helper(int[] num, boolean[] used, ArrayList<Integer> item,
ArrayList<ArrayList<Integer>> res){
    if(item.size() == num.length){
        res.add(new ArrayList<Integer>(item));
        return;
    }
    for(int i = 0; i < num.length; i++){
        if(i > 0 && num[i] == num[i - 1] && !used[i - 1]){
            continue;
        }
        if(!used[i]){
            used[i] = true;
            item.add(num[i]);
            helper(num, used, item, res);
            item.remove(item.size() - 1);
            used[i] = false;
        }
    }
}
```

做题时的感悟:

1. 这种避免重复的机制有点难理解，下面给出一种合理的解释:

假设有两个1，排序后位置不同，我们规定这两个1在排序中出现的顺序必须和数组中位置顺序一样，也就是第一个1只能出现在前面，第二个1只能出现在后面，这样就消除了重复

解。对应到代码中，要排除的情况就是在前面的位置选择第二个1，这时检查发现第一个1还没用过就是这种情况，于是可以直接跳过了。

2. 对于这种有重复元素的题，处理重复元素的必要条件是数组中元素有序，所以先对数组进行排序的必须的。

5. Combinations，这道题是NP问题的题目，时间复杂度没办法提高，用到的还是N-Queens中的方法：用一个循环递归处理子问题。实现的代码跟Combination Sum非常类似而且更加简练，因为不用考虑重复元素的情况，而且元素只要到了k个就可以返回一个结果。套用黄金模板，稍作修改即可。

代码如下：

```
// Solution 1 - Classic Model
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(n <= 0 || k <= 0 || n < k){
        return res;
    }
    ArrayList<Integer> item = new ArrayList<Integer>();
    helper(n, k, 1, item, res);
    return res;
}

private void helper(int n, int k, int start, ArrayList<Integer> item, ArrayList<ArrayList<Integer>> res){
    if(item.size() == k){
        res.add(new ArrayList<Integer>(item));
        return;
    }
    for(int i = start; i <= n; i++){
        item.add(i);
        helper(n, k, i + 1, item, res);
        item.remove(item.size() - 1);
    }
}
```

6. Combination Sum，这个题是一个NP问题，方法仍然是N-Queens中介绍的套路，也仍然使用模板。基本思路是先排好序，然后每次递归中把剩下的元素一一加到结果集合中，并且把目标减去加入的元素，然后把剩下元素（包括当前加入的元素）放到下一层递归中解决子问题。算法复杂度因为是NP问题，所以自然是指数量级的。注意在实现中for循环中第一步有一个判断，那个是为了去除重复元素产生重复结果的影响，因为在这里每个数可以重复使用，所以重复的元素也就没有作用了，所以应该跳过那层递归。

```
// Solution 1 – Classic Model
public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates, int target)
{
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(candidates == null || candidates.length == 0){
        return res;
    }
    Arrays.sort(candidates);
    ArrayList<Integer> item = new ArrayList<Integer>();
    helper(candidates, 0, target, item, res);
    return res;
}

private void helper(int[] candidates, int start, int target, ArrayList<Integer> item,
ArrayList<ArrayList<Integer>> res){
    if(target < 0){
        return;
    }
    if(target == 0){
        res.add(new ArrayList<Integer>(item));
        return;
    }
    for(int i = start; i < candidates.length; i++){
        if(i > 0 && candidates[i] == candidates[i - 1]){
            continue;
        }
        item.add(candidates[i]);
        helper(candidates, i, target - candidates[i], item, res);
        item.remove(item.size() - 1);
    }
}
```

做题时的感悟:

1. 由于target - num[i]可能产生小于0的结果，所以要单独判断一下这种情况。其实这种情况就是不符合条件，直接返回即可。

```
if(target < 0){
    return;
}
```

2. 注意循环中判断重复的条件:

```
if(i > 0 && candidates[i] == candidates[i - 1]){
    continue;
}
```

以及递归中下一个元素的取用:

```
helper(candidates, i, target - candidates[i], item, res);
```

之所以如上面这样写代码，是因为题目中要求: The same repeated number may be chosen from C unlimited number of times. 也就是说1个元素可以被取用无限多次，所以后面跟这个元素相同的元素一概没有用了。所以递归调用时，要代入i而不是i + 1，就是不断使用这个元素直到不能再用了为止。之后，只要i > 0时遇到该元素，直接跳过即可。

7. Combination Sum II，这道题跟Combination Sum唯一的区别就是这个题目中单个元素用过就不可以重复使用了。乍一看好像区别比较大，但是其实实现上只需要一点点改动就可以完成了，就是递归的时候传进去的index应该是当前元素的下一个。

在这里我们还是需要在每一次for循环前做一次判断，因为虽然一个元素不可以重复使用，但是如果这个元素重复出现是允许的，但是为了避免出现重复的结果集，我们只对于第一次得到这个数进行递归，接下来就跳过这个元素了，因为接下来的情况会在上一层的递归函数被考虑到，这样就可以避免重复元素的出现。

代码如下:

```
// Solution 1 – Classic Model
public ArrayList<ArrayList<Integer>> combinationSum2(int[] num, int target) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length == 0){
        return res;
    }
    Arrays.sort(num);
```

```
        ArrayList<Integer> item = new ArrayList<Integer>();
        helper(num, 0, target, item, res);
        return res;
    }

    private void helper(int[] num, int start, int target, ArrayList<Integer> item,
ArrayList<ArrayList<Integer>> res){
        if(target < 0){
            return;
        }
        if(target == 0){
            res.add(new ArrayList<Integer>(item));
            return;
        }
        for(int i = start; i < num.length; i++){
            if(i > start && num[i] == num[i - 1]){
                continue;
            }
            item.add(num[i]);
            helper(num, i + 1, target - num[i], item, res);
            item.remove(item.size() - 1);
        }
    }
```

做题时的感悟:

1. 在判断target < 0的同时，可以判断一下start >= num.length，即start已经超出数组的下标了，直接返回即可。

```
    if(target < 0 || start >= num.length){
            return;
    }
```

2. 注意循环中判断重复的条件:

```
        if(i > start && num[i] == num[i - 1]){
                continue;
        }
```

以及递归中下一个元素的取用:

```
        helper(num, i + 1, target - num[i], item, res);
```

之所以如上面这样写代码，是因为题目中要求: Each number in C may only be used once in the combination. 也就是说1个元素只可以被取用1次，所以后面跟这个元素相同的元素可能还会被用到，因为1个当前元素可能不够用。递归调用的时候，要代入i + 1，就是继续尝试下一个元素。之后，必须i > start时遇到该元素，才可以被跳过，因为情况会被包含在前面的那次递归中。但如果i > 0（错误）时就跳过，很可能需要多个某元素，但遇到就直接跳过了那个元素，导致错误的结果。

8. Letter Combinations of a Phone Number，这道题比较简单，直接套用模板即可。依次读取数字，然后把数字可以代表的字符依次加到当前字符串中，然后递归剩下的数字串，得到结果后加上自己返回回去。假设总共有n个digit，每个digit可以代表k个字符，那么时间复杂度是O(k^n)，就是结果的数量，空间复杂度也是一样。

```
// Solution 1 – Classic Model
public ArrayList<String> letterCombinations(String digits) {
    ArrayList<String> res = new ArrayList<String>();
    if(digits == null || digits.length() == 0){
        res.add("");
        return res;
    }
    StringBuilder item = new StringBuilder();
    helper(digits, 0, item, res);
    return res;
}

private void helper(String digits, int index, StringBuilder item, ArrayList<String> res){
    if(index == digits.length()){
        res.add(item.toString());
        return;
    }
    String str = getString(digits.charAt(index));
    for(int i = 0; i < str.length(); i++){
        item.append(str.charAt(i));
        helper(digits, index + 1, item, res);
        item.deleteCharAt(item.length() - 1);
    }
}
```

```
private String getString(char c){
    switch(c){
        case '2':
            return "abc";
        case '3':
            return "def";
        case '4':
            return "ghi";
        case '5':
            return "jkl";
        case '6':
            return "mno";
        case '7':
            return "pqrs";
        case '8':
            return "tuv";
        case '9':
            return "wxyz";
        case '0':
            return " ";
        default:
            break;
    }
    return "";
}

// Solution 2 – Iteration
public ArrayList<String> letterCombinations(String digits) {
    ArrayList<String> res = new ArrayList<String>();
    res.add("");
    if(digits == null || digits.length() == 0){
        return res;
    }
    for(int i = 0; i < digits.length(); i++){
        ArrayList<String> newRes = new ArrayList<String>();
        String letters = getLetters(digits.charAt(i));
        for(int j = 0; j < res.size(); j++){
            String cur = res.get(j);
            for(int k = 0; k < letters.length(); k++){
                newRes.add(cur + letters.charAt(k));
```

```
                }
            }
            res = newRes;
        }
        return res;
    }
```

做题时的感悟：

1. switch中可以直接用char进行判断，因为char型可以自动转换成int型，所以不用强制把char转换成int在放入switch中判断。

2. 在迭代方法中，要先在res中添加一个空字符串，不然res的size为0，会直接跳过循环。