链滴

# 回溯算法解决子集、组合和排列问题

作者：vcjmhg
原文链接：https://ld246.com/article/1619773723151
来源网站：链滴

# 概述

之前《"回溯法套路总结与应用"》这篇文章中，讲了一个回溯算法的通用模板：

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return
    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

但我们在套用该模板解决实际算法问题时会发现，针对不同类型的问题，会有不同的细节需要注意。而本篇文章就以**子集**、**组合**、**排列**三类常见搜索问题为例，来讲解使用回溯法来解决这三类问题所需考虑的不同细节和思路。

# 子集

问题描述很简单：

在使用回溯法解决问题之前，我们可以根据题意画出搜索过程中产生的回溯树。

我们以**示例1**为例：

首先，空集[]肯定是一个子集

然后，以1开头的子集：[1],[1,2],[1,3],[1,2,3]。

以2开头的子集：[2],[2,3]

以3开头的子集：[3]。

然后将上述搜索过程写成一个回溯树：

从回溯树中可以看到，每次在做出一个选择（分支）时，都是添加一个新的未被使用的元素放到搜索表中。但我们此时会有一个问题，如果来保证搜索时的元素是不重复的呢？

实际上，我们只需要添加一个start参数来控制递归，每次通过for循环来做出选择的时候，都从start开始遍历。

具体代码如下：

```
class Solution {
    List<List<Integer>> res = null;
    public List<List<Integer>> subsets(int[] nums) {
        res = new LinkedList<>();
        LinkedList<Integer> trace = new LinkedList<>();
        backtrack(nums, 0, trace);
        return res;
    }
    private void backtrack(int[] nums, int start, LinkedList<Integer> trace) {
        //结束条件判断
        res.add(new LinkedList<>(trace));
    //通过start来控制选择的起始点
        for(int i = start; i < nums.length; i++) {
            //做出选择
            trace.add(nums[i]);
            //递归回溯
            backtrack(nums, i + 1, trace);
            //撤销选择
            trace.removeLast();
        }
    }
}
```

# 排列

组合问题，题目描述如下：

同样的，我们需要根据题意画出对应的回溯树。

以n = 4, k = 2为例，

首先，以1开头，长度为2的组合有:[1,2],[1,3],[1,4]。

以2的开头的组合有：[2,3],[2,4]。

以3开头的组合有：[3,4]。

画出对应的回溯树：

从回溯树中，我们可以看到组合问题最终搜索到的结果长度是一定的都等于k，因而回溯结束的条件定可以用k == trace.size()来进行判断，当相等时，将搜索到的trace的结果添加到res列表中。另一面，每一轮的搜索范围也不是所有元素，而是每次搜索添加的元素都是**新的未被添加的元素**，因而考增加一个start参数，来控制搜索的起始范围。

具体实现代码如下：

```
class Solution {
    List<List<Integer>> res = null;
    public List<List<Integer>> combine(int n, int k) {
        res = new LinkedList<>();
        LinkedList<Integer> trace = new LinkedList<>();
        backtrack(n, 1, k, trace);
        return res;
    }
    private void backtrack(int n, int start, int k, LinkedList<Integer> trace) {
        //当trace中节点数量与k相同时，则记录结果并停止搜索
        if(k == trace.size()) {
            res.add(new LinkedList<>(trace));
            return;
        }
        for(int i = start; i <= n; i++) {
            //做出选择
            trace.add(i);
            //回溯
            backtrack(n,i + 1, k, trace);
            //撤销选择
            trace.removeLast();
        }
    }
}
```

执行结果如下：

# 组合

组合问题描述如下：

与上述过程类似，根据题意画出其回溯树。

我们以题中例子[1,2,3]为例，

以1开头的排列为：[1,2,3],[1,3,2]

以2开头的排列为：[2,1,3],[2,3,1]

以3开头的排列为：[3,1,2],[3,2,1]

对应的回溯树如下所示：

组合问题和前边排列问题的不同点在于，组合问题选择时，选择的范围会更广，每次做选择的时候，了trace中已经存在的元素之外，其他的元素都要选择一遍。因而在解决该问题时，在"做出选择"之前要判断当前选择的元素是否在trace中已经存在，如果存在则放弃此次选择。具体代码如下：

```
class Solution {
    List<List<Integer>> res = null;
    public List<List<Integer>> permute(int[] nums) {
        res = new LinkedList<>();
        LinkedList<Integer> trace = new LinkedList<>();
        backtrack(nums, trace);
        return res;
    }
    private void backtrack(int[] nums, LinkedList<Integer> trace) {
        //搜寻到结果集等于nums.size()则证明当前的一个全排列搜索解决完成
        if(nums.length == trace.size()) {
            res.add(new LinkedList<>(trace));
            return;
        }
        for(int num : nums) {
            //如果选择的元素已经存在
            if(trace.contains(num)) continue;
            //做出选择
            trace.add(num);
            backtrack(nums,trace);
            //撤销选择
            trace.removeLast();
        }
    }
}
```

运行结果如下：

# 总结

本文主要讲了通过回溯法如何解决排列、组合、子集这三类问题的基本思路：

组合问题关键点在于用一个start来保证每次选择的元素是之前未被选择过的

排列问题关键点在于通过contains()来保证每次选择的元素都未被包含在trace中

这两类问题回溯结束的时机都是搜索到的元素达到了预定的长度，**即我们可以判断trace中元素的长来判断是否终止此次回溯。**

而子集问题则不然，因为它的长度是变长的，所以每次进入搜索的第一件事情就是将结果加入到结果中。

# 参考

1. 《labuladong的算法小抄》