递归 Recursion

无隅

从一个故事说起。。。

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？

从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……

什么是递归

- 在数学和计算机科学中，递归指由一种（或多种）简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况。
- 递归指在函数的定义中使用函数自身的方法

如何解决问题

- 拆解成更小的问题
- 求解小问题
- 利用小问题的结果解决原来的问题
- 如果小问题与原问题相似只是规模不同，那么这就是递归问题

递归三要素

- 拆解寻找子问题（得到递归规则）
- 最小子问题（基本问题）

  解决最小子问题是指可以直接得到答案问题并不需递归计算
- 递归终止退出条件

数学归纳

- 证明一个给定的陈述
- 数学归纳法对解题的形式要求严格，数学归纳法解题过程中，
- 第一步：验证 n 取第一个自然数时成立
- 第二步：假设 n=k 时成立，然后以验证的条件和假设的条件作为论证的依据进行推导，在接下来的推导过程中不能直接将 n=k+1 代入假设的原式中去。
- 最后一步总结表述

递归的代码问题

- 基本情况
- 递归规则
- 用编码函数表示问题
  - 定义基本参数

    定义问题的参数
  - 定义返回值

面试中简单递归问题

斐波那契数列

有这样的一个数列： 1, 1, 2, 3, 5, 8, 13

这个数列从第 3 项开始，每一项都等于前两项之和。

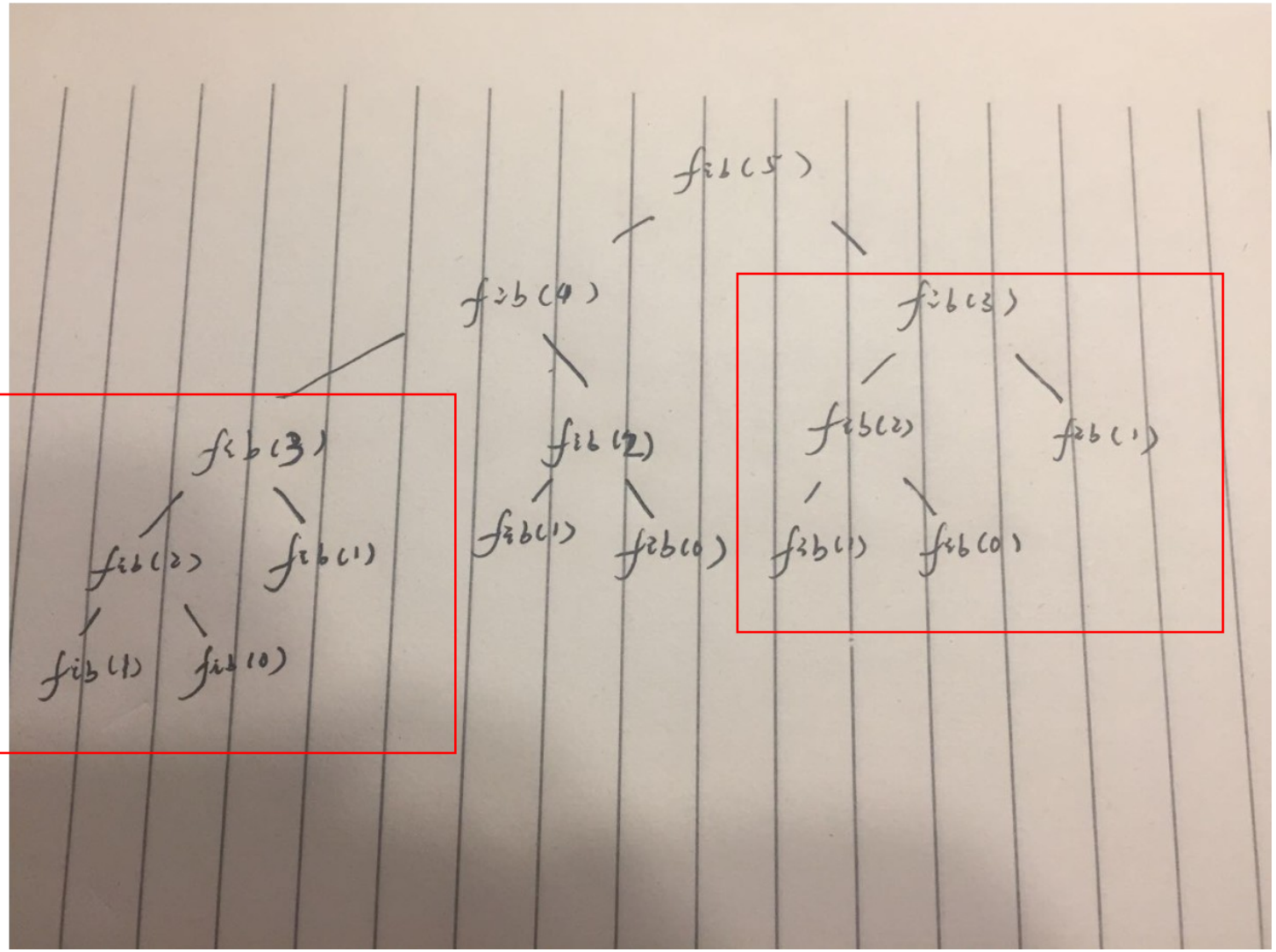
求第 n 项的值

斐波那契数列

1, 1, 2, 3, 5, 8, 13

- 基本情况： $F(0) = 0,\ F(1) = 1,\ \mathrm{F}(2) = 1$
- 递归规则: $F(n) = F(n\ -1) + F(n\ -2)$

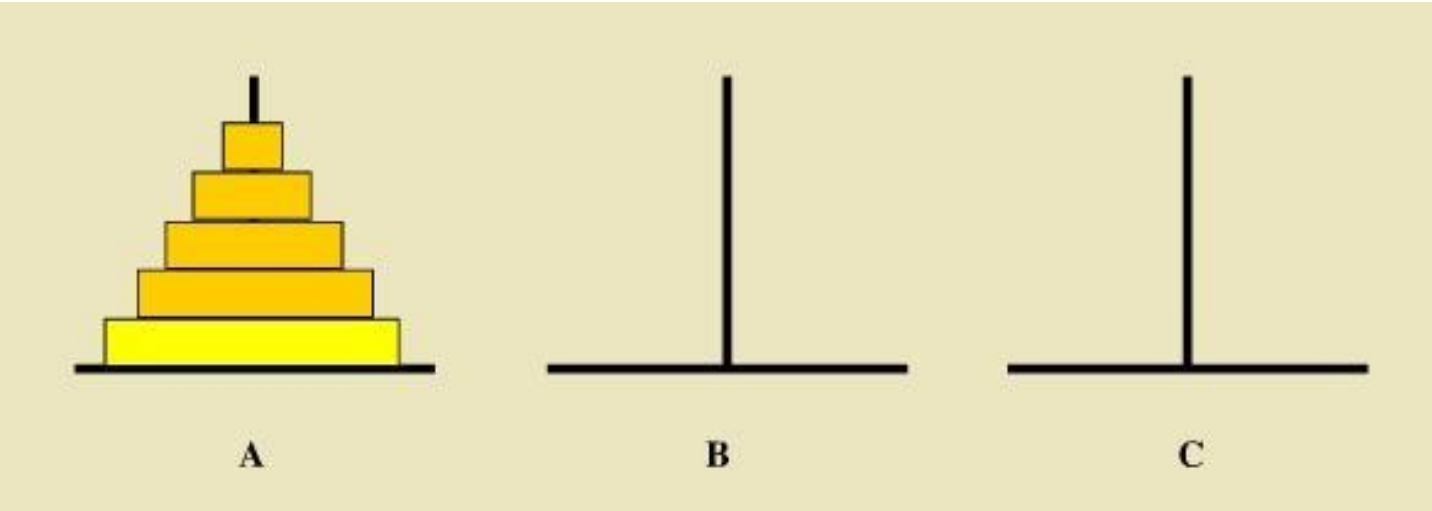
```
// Fibonacci calculation
int Fibonacci (int n) {
    //Base Case
    if(n == 0) return 0;
    if(n == 1) return 1;
    //recursion rule
    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(1)
fib(0)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(1)
fib(0)

汉诺塔

有三根杆子 A ， B ， C 。 A 杆上有 N 个 (N>1) 穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至 C 杆：

1. 每次只能移动一个圆盘；
2. 大盘不能叠在小盘上面。



汉诺塔

- 基本情况： n=1，直接将盘子从 A 移到 C
- 递归规则：

1. 将 A 上的前 n-1 个盘子从 A 移到 B
2. 将第 n 个盘子，也就是最底下的盘子从 A 移到 C
3. 将 B 上剩下的 n-1 个盘子从 B 移到 C

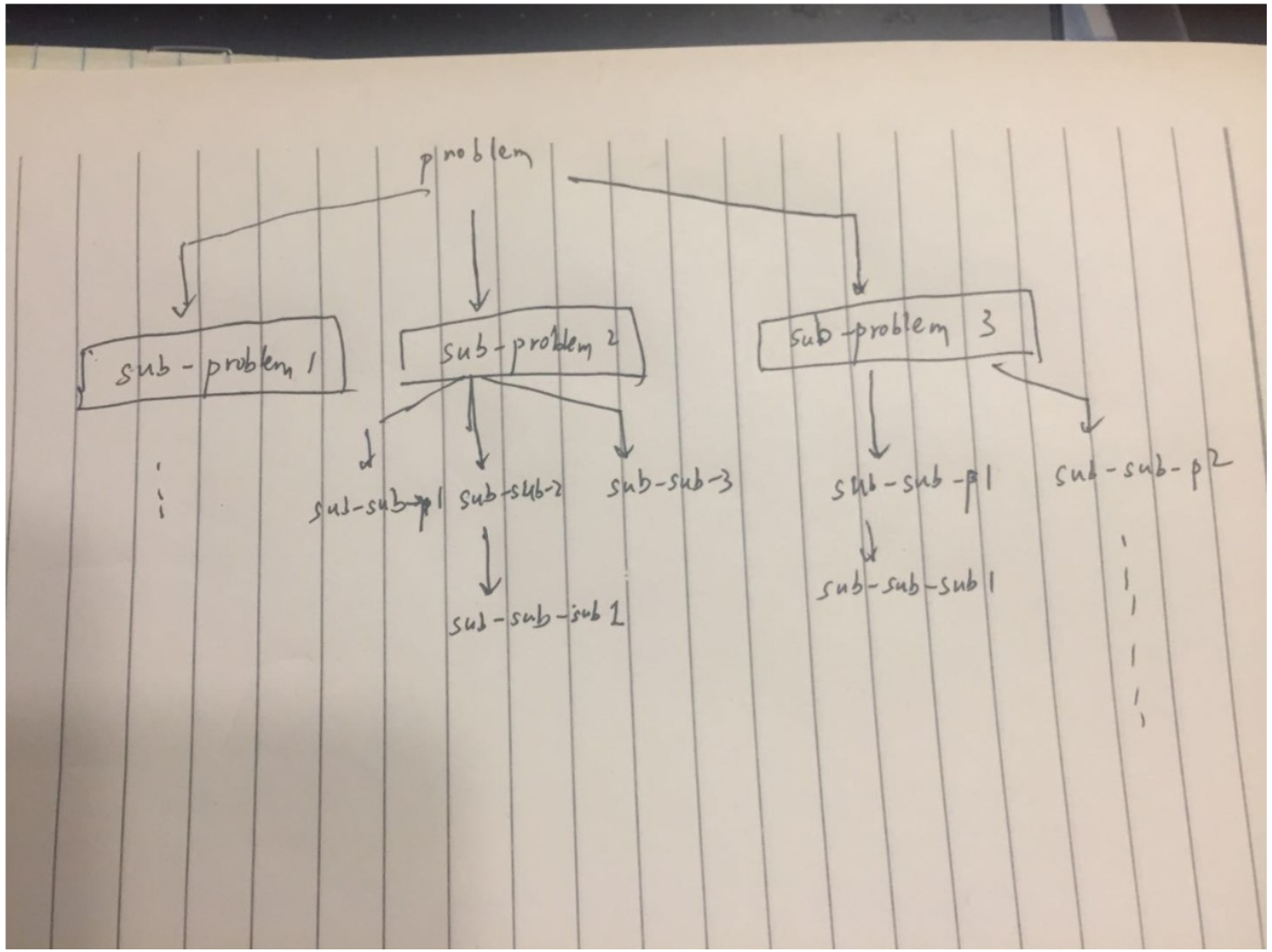
- 递推函数：

MoveHanoi(int n, char origin, char destination, char buffer)

```java
public void MoveHanoi(int n, char origin, char destination, char buffer) {
    if (n == 1) {
        System.out.println("Move Step: " + origin + " to " + destination);
        return;
    }
    MoveHanoi(n - 1, origin, buffer, destination);
    System.out.println("Move " + origin + " to " + destination);
    MoveHanoi(n - 1, buffer, destination, origin);
}
```

递归经典问题 – 回溯法 backtracking

递归经典问题 – 回溯法 backtracking



递归经典问题 – 回溯法 backtracking

- 回溯法是一种选优搜索法
- 按选优条件向前搜索，以达到目标
- 但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法
- 满足回溯条件的某个状态的点称为“回溯点”

回溯法的想法

- 在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去，如果该结点不包含问题的解，则逐层向其祖先结点回溯。（其实回溯法就是对隐式图的深度优先搜索算法）。
- 若用回溯法求问题的所有解时，要回溯到根，且根结点的所有可行的子树都要已被搜索遍才结束。
- 而若使用回溯法求任一个解时，只要搜索到问题的一个解就可以结束。

回溯法步骤

- 针对所给问题，确定问题的解空间：首先应明确定义问题的解空间，问题的解空间应至少包含问题的一个（最优）解。
- 确定结点的扩展搜索规则
- 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

回溯法模板 – 以 Subsets 为例

给定一组不含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集

输入 : nums = [1,2,3]

输出 : [[3], [1], [2], [1,2,3], [1,3], [2,3], [1,2], []]

https://leetcode-cn.com/problems/subsets/description/

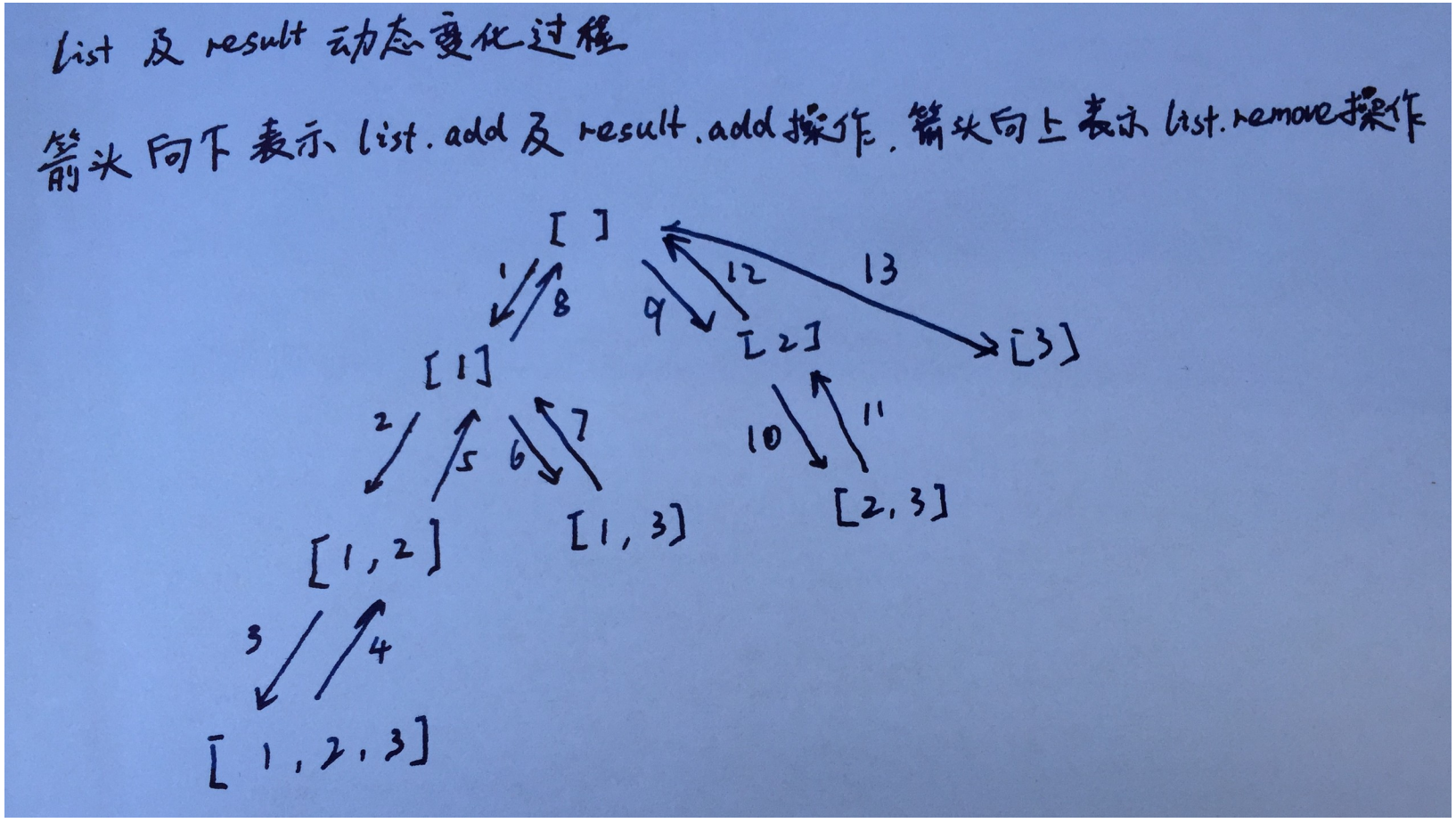
回溯法模板 – 以 Subsets 为例

```java
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    if (nums == null || nums.length == 0){
        return result;
    }
    List<Integer> list = new ArrayList<Integer>();
    Arrays.sort(nums);
    subsetHelp(result, list, nums, 0);
    return result;
}

private void subsetHelp(List<List<Integer>> result, List<Integer>list, int[] nums, int pos){
    result.add(new ArrayList<Integer>(list));

    for (int i = pos; i < nums.length; i++){
        list.add(nums[i]);
        subsetHelp(result, list, nums, i + 1);
        list.remove(list.size() - 1);

    }
}
```

很多题目的解法均是在这个模板的基础上，但剪枝的条件会有所不同

回溯法模板 – Subsets 函数调用分析

backtracking 可用图示与函数运行的堆栈来理解，以 [1, 2, 3] 为例，
下图为 list 及 result 的变化过程，箭头向下表示 list.add 及 result.add 操作，
箭头向上表示 list.remove 操作



子集 II

给定一个可能包含重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

说明：解集不能包含重复的子集

输入 : [1,2,2]

输出 : [[2], [1], [1,2,2], [2,2], [1,2], []]

https://leetcode-cn.com/problems/subsets-ii/description/

子集 II – 分析

- 与 subsets 模板基本一致
- 既然要求 Unique 的，就想办法排除重复的
- 初始化 subset: {}
- 添加 “ 1”: {}, {1}
- 添加 “ 2”: {}, {1}, {2(1)}
- {1, 2(1)}, {1, 2(2)} 是重复的 , {1,2(1),2(2)}, {1,2(2),2(3)} 也是重复的
- 结论： 我们只关心取多少个 2 ， 不关心取哪几个

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

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

    Arrays.sort(nums);
    List<Integer> list = new ArrayList<Integer>();
    helper(result, list, nums, 0);
    return result;
}
private void helper(List<List<Integer>> result, List<Integer> list, int[] nums, int pos) {
    result.add(new ArrayList<Integer>(list));
    for (int i = pos; i < nums.length; i++) {
        if ( i != pos && nums[i] == nums[i - 1]) {
            continue;
        }
        list.add(nums[i]);
        helper(result, list, nums, i + 1);
        list.remove(list.size() - 1);
    }

}
```

全排列

给定一个没有重复数字的序列，返回其所有可能的全排列。

示例：

输入：[1,2,3] 输出：[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]

全排列

- 与 subsets 模板基本一致
- 递归地将元素添加到一个列表
- 直到所有元素被添加。确保列表不包含当前元素
- 注意：这种类型的排列问题总是需要添加 / 删除之前 / 之后递归

```java
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    if (nums == null && nums.length == 0){
        return result;
    }

    List<Integer> list = new ArrayList<Integer>();
    permuteHelper(result, list, nums);
    return result;
}
private void permuteHelper(List<List<Integer>> result, List<Integer> list, int[] nums){
    if (list.size() == nums.length) {
        result.add(new ArrayList<Integer>(list));
    }

    for(int i = 0; i < nums.length; i++) {
        if (list.contains(nums[i])) {
            continue;
        }
        list.add(nums[i]);
        permuteHelper(result, list, nums);
        list.remove(list.size() - 1);
    }
}
```

组合总和

给定一个无重复元素的数组 candidates 和一个目标数 target ，找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以无限制重复被选取。

说明：

所有数字（包括 target）都是正整数。

解集不能包含重复的组合。

示例 1:

输入: candidates = [2,3,6,7], target = 7, 所求解集为: [[7], [2,2,3]]

示例 2:

输入: candidates = [2,3,5], target = 8, 所求解集为: [[2,2,2,2], [2,3,3], [3,5]]

```java
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    if (candidates.length == 0 || candidates == null) {
        return null;
    }
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> list = new ArrayList<Integer>();
    Arrays.sort(candidates);
    helper(result, list,candidates, target, 0);
    return result;
}

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

    for (int i = pos; i < candidates.length; i++) {
        list.add(candidates[i]);
        helper(result, list, candidates, target - candidates[i],i);
        list.remove(list.size() - 1);
    }
}
```

分割回文串

给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。

返回 s 所有可能的分割方案

示例：

输入：“aab” 输出：[["aa","b"], ["a","a","b"]]

https://leetcode-cn.com/problems/palindrome-partitioning/description/

```java
public List<List<String>> partition(String s) {
    ArrayList<List<String>> result = new ArrayList<List<String>>();
    ArrayList<String> list = new ArrayList<String>();
    helper(result, list, s, 0);
    return result;
}
private void helper(List<List<String>> result, List<String> list,String s, int pos) {
    if (pos == s.length()) {
        result.add(new ArrayList<String>(list));
        return;
    }
    for (int i = pos + 1; i <= s.length(); i++) {
        String prefix = s.substring(pos,i);
        if (!isPalindrome(prefix)) {
            continue;
        }
        list.add(prefix);
        helper(result, list, s, i);
        list.remove(list.size() - 1);
    }
}
private boolean isPalindrome(String str) {
    if (str == null) {
        return false;
    }
    int start = 0;
    int end = str.length() - 1;
    while (start < end) {
        if (str.charAt(start) != str.charAt(end)) {
            return false;
        }
        start++;
        end--;
    }
    return true;
}
```

括号生成

给出 n 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且有效的括号组合。

例如，给出 n = 3，生成结果为：

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

https://leetcode-cn.com/problems/generate-parentheses/description/

```java
public List<String> generateParenthesis(int n) {
    ArrayList<String> result = new ArrayList<String>();
    if (n <= 0) {
        return result;
    }
    char[] str = new char[2 * n];
    helper(result, str, n, n, 0);
    return result;
}
private void helper(ArrayList<String> result, char[] str, int leftRemind, int rightRemind, int index) {
    if (index == str.length) {
        String s = new String(str);
        result.add(s);
    }
    if (leftRemind > 0) {
        str[index] = '(';
        helper(result, str, leftRemind - 1, rightRemind, index + 1);
    }
    if (rightRemind > leftRemind) {
        str[index] = ')';
        helper(result, str, leftRemind, rightRemind - 1, index + 1);
    }
}
```

回溯法总结

- 在搜索空间中尝试：

1. 解被找到
2. 没有更有意义的路径可以尝试（没有更多的搜索空间）

- 将第 N 层的问题分解成 M 个第 N + 1 层的子问题

递归总结

- 递归是一种策略，一种思想
- 递归框架（递归三要素）：

1. 拆解寻找子问题（得到递归规则）
2. 解决最小子问题（基本问题）

   解决最小子问题是指可以直接得到答案问题并不需递归计算

3. 递归终止退出条件

- 总是试图将问题分解成子问题，首先解决的子问题