Leetcode:

对于两个数的和的问题，在要求时间复杂度的情况下，通过空间换时间的方法，一定要利用哈希解决。

**Array**:（注意属猪边界）

1. 对于用后面元素覆盖前面元素的问题，我可以用不满足作为if的条件，这是两个索引都在递增，进行自己对自己的覆盖；而满足条件的时候，一个索引是不变的（满足题意），而另一个索引在找后面将要覆盖的元素；
2. 对于数组有序或者近似有序的情况要用到二分查找；
3. 两个索引只遍历一遍大多说情况会被用到（解决）。
4. 对于有序数组合并的问题，只需要在合并的时候只需要关注小的元素

**String:**

1. 对于判读类型的字符串为题可以借助外界变量(可能多个外界变量)
2. 对于字符串子序列的问题或者配准的问题一般都考虑dp解决；当我们要穷举所有的情况的时候，我们通常考虑递归(回溯法）

**Tree:（注意节点判空）**

1. 对于二叉树一般都会用到递归解决（基本通用）；在其他可能会用到栈（stack）这种结构,也可以借助其他数据结构。如果要用到两个栈，可以使用双端队列。
2. 在求解特定左子树或者右子树的时候，我们用该节点的父节点为当前节点；
3. 在求解特定值是可以借助临时变量或者数组，然后用中序遍历（前或者后）；
4. 大多数树的操作都是涉及到其遍历（递归方式）

**List:（注意节点判空）**

1. 对于有序链表和并的问题，只需要在合并的时候只需要关注小的元素
2. 对于链表合并和反转的递归方式（不是很清楚）。
3. 对于链表在一次遍历完成，一般用两个指针。
4. 对于链表的反转（移动链表节点），可以借助一个额外的头结点或者用二级指针,注意反转后的最后一个节点置空（原来的第一个节点），以及对于循环判断是第一个节点为空（一般为第一个）还是第二个。

**Math:**

DP:

对于保存子问题解的空间变量，对于不同的问题其数组或者单个变量也不同。

在运用动态规划是要利用子问题的解，所以一个合理的遍历顺序会保证前面的解都被计算出来了,边界条件和状态转移方程,dp的一个过程：

（1）划分阶段：

（2）确定状态：dp[i][j]（代表s[:i-1]和p[:j-1]是否match）

（3）边界条件：

A. dp[0][0] = True (代表两个空数组match)

B. dp[i][0] = False (代表s为空，p不为空则必然不匹配)

C. dp[0][j] = dp[0][j-2] if p[j-1]=='\*' and j>=2

（4）状态转移方程：(分情况考虑)

A. p[j] != '\*' 时，

if p[j] != '\*' :

dp[i+1][j+1] = dp[i][j] and (s[i] == p[j] or p[j] == '.' )

B. p[j] == '\*' : （重点考虑该情况）

if p[j] == '\*' :

a. 该 '\*' match了s中的0个字符：【注：该处（及下方）提到的match了x个字符都是就当前而言的，不考虑s及p的第i、j之后的字符match情况】

dp[i+1][j+1] = dp[i+1][j-1]

b. 该 '\*' match了s中的1个字符：

dp[i+1][j+1] = dp[i][j-1] and(s[i] == p[j-1] or p[j-1] == '.' )

c. 该 '\*' match了s中的多个字符：

dp[i+1][j+1] = dp[i][j+1] and (s[i] == p[j-1] or p[j-1] == '.' )

有了边界条件和状态转移方程，实现就变得十分简单了

递归：递归分为递（入栈）和归（出栈），对于每一个子问题，有得实在入栈的过程中进行操作，到终止条件就结束；有的是到终止条件才开始，在出栈的过程中进行操作。

**Other:**

对于列出所有结果的问题，一般优先考虑递归，基于回溯的思想。回溯是指当前情况下有n中选择，对于n中情况我们都尝试每一种，当该种情况不可用的时候（不满足条件），我们就返回false，回退到失败的上一层（？）；如果成功则将其添加进去；

**Dfs**：