**图：**

1. **“最少”**：最少路径，最少…… 一般联想到 图，**BFS**，回溯法，来寻找最短路径；
2. **中序遍历：**

二叉搜索树的中序遍历是一个严格递增序列；可以通过这个特点来判断一个二叉树是否为二叉搜索树，但还要注意的是，二叉搜素树的中序遍历结果是一个严格递增数列，不存在同值的情况；所以需要将结果 通过集合set() 来去重。

3.

**数组：**

**1．哈希表** K数之和  
K数之和，一般是先确定两数之和，将其保存在哈希表(字典)中，各种和的个数，或者下标；  
然后对应另外的k-2个数，使用循环遍历，  
**不同值 不同下标**  
**不同值**：要先排序，在循环时，遇到和上一个值相同的情况，跳过(右移); 同时，要注意，当对k-2个数从左往右循环时，对应 哈希表 的建立，应该从右往左循环，这样当遇到同值情况时，会先保存下标靠右的；这样，在最后组合时，满足 哈希表中的数字都在 k-2个数的右边，不会漏解(如果求哈希表时，从左往右，就难满足最后组合的要求，在有同值的情况下，容易漏解。) e.g. 本题  
**不同下标**：不用排序，直接找到满足和的组合即可。

**2.对于圆环这种数组，要想循环访问，就要注意左右边界：**

左边界小于0时，不是没有值，而是到达了末尾，及右边界；

右边访问大于有边界时，再往右，是到 开头；

所以可以通过 增加取余的操作来保证不超过边界

Pre = (i-1+n)%n pos = (i+n)%n

**动态规划：**

1. 一般状态转移方程的情况时，当前状态的最优值 有前几个状态决定，**每一种状态的选取都有对应的条件**，比如背包问题中，选了这个物品，那么对应的上一个的状态就是物品数量少1个，背包容量比当前容量少一个背包的容量时的状态最优值。

树形DP也是这种情况，

1. 问题形式
2. **最优，最多，最少**等最值问题，典型的动态规划题
3. 多少种情况，多少种路径，一般可以用动态规划，也可以尝试回溯法，找到所有的

情况，而不仅仅是数量；乍一看没有上面的直观要用动态规划，

多少种情况：从低到上，根据已知的，逐渐扩大规模；

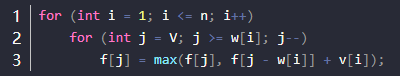
比如：不同路径总数 leetcode 62、63

1. **背包问题**
2. 0-1背包

常规的动态规划题，每一件物品在上一个状态的基础上可选 可不选，



优化空间复杂度为o(n)



内层循环，**从右往左，**为了保证每件物品只取一次，

换种说法，如果从左往右，则每次会更新左边的f[j]，这个有可能选取了物品i，

但是如果后面要考虑这个f[j]时，就会导致重复选物品i，就违反了物品i只选一次的要求。

1. 完全背包

每一种商品的数量无限，上来可能通过性价比高低的方式来选择物品，即**贪心**，但不行，

反例 A：w = 5,v= 5 B:w = 7 v= 8, V = 10，显然B的性价比高，但是选择了B只能选择一个，价值为8；但如果选择两个A，则背包装满，价值为10，最优；

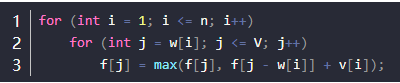
第i种商品，可以选择0、1、2，……，k（k\*w[i]<=V）,

方法一：使用二进制的方法，类似于下面重复背包，复杂度为o(V )

方法二：复杂度o(VN)，状态转移方程使用一维数组，类似于0-1背包中的情况，不过第二层循环，不是之前的从右往左，而是**从左往右**，

从右往左，是为了保证第i件物品只选择一次；

从左往右，认为，每一件物品不止取一次，后面可能会在之前取一次或多次的基础上再取一次第i件物品，所以是从左往右。



1. 重复背包

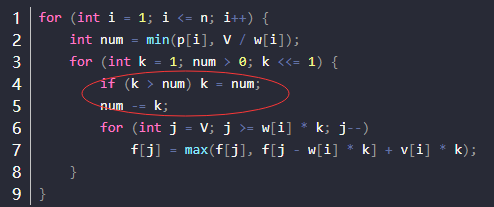
**前提**：从二进制的角度考虑，对应整数n，[1,n]范围内的任意一个整数，都可以写成k个二进制数及尾数m之和的形式。1，2，4，2^(k-1) ,m (m+2^k-1 = n)

因此，可以把第i种物品，有p[i]件，按上面的形式分成k件独立的物品，相应的重量和价值也和上面的数对应（系数即倍数）。

从而转化成基础的0-1背包问题。

细节：

1. k = min(p[i],V/w[i])，即两者的较小者；
2. 一共有三层循环，第一层是商品，第二层是k的循环，第三层是容量的循环，从w[i]\*k开始往右循环；
3. 图中红色圈出来的部分，当k超过2^(k-1)时，k就要取尾数了。



**单源最短路径问题：**

**Dijkstra**算法：**贪心**算法

**Floyd**算法：**动态规划**

**最小生成树问题**

不存在圆环的树，树上边的权重之和最小。

**Prim算法：**从**顶点**出发，搜索生成树上的点到剩余点之间的最小权重，

**kruskral算法：**从**边**出发，先将边上的权值从小到大排序

**AVL(自平衡二叉搜索树)**：

每个节点左右子树的高度差不超过1

**删除：T(n) = o(logn)** 通过把要删除的节点向下旋转成一个叶子节点，接着直接删除这个叶子节点来完成，旋转过程中最多有logn个节点被旋转，而每次AVL旋转耗费恒定的时间，所以删除的时间复杂度 o(logn)

**查找：T(n) = o(logn)**

**构建：**

1. 左-左型：做右旋；
2. 右-右型：做左旋；
3. 左-右型：先做左旋，后做右旋；
4. 右-左型：先做右旋，再做左旋。