1. 链表与数组：

读取 插入 删除

数组 O(1) O(n) O(n)

链表 O(n) O(1) O(1)

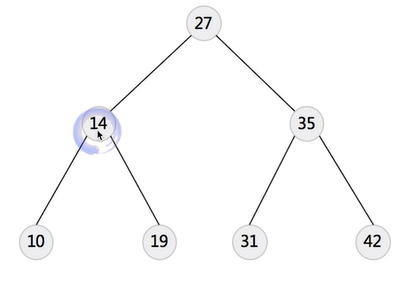
问题：

1. 链表的反转
2. 树 二叉树（Binary Tree）二叉搜索树（Binary Search Tree）图（Graph）

树：每个节点有多个指针指向后继节点

二叉树：每个节点最多只有两个后继节点 （每个节点都严格有两个子节点的二叉树叫完全二叉树）

二叉搜索树： 左子树所有点小于根节点 右子树所有点大于根节点 且递归地左右子树也是二叉搜索树（查找时每次与根节点比较能减少一半的数据量log2 N）



拓展：红黑树，AVL树 c++ java底层二叉树都用红黑树实现

问题：

1. 验证二叉搜索树 98

给定一棵树的根节点指针 判断该树是否是二叉排序树。

解一：根据二叉搜索树特性 中序遍历得到的数是升序的。先中序遍历该树，再判断的得到的数组是否升序。也可以不用先得到该数组，直接每次判断当前数是否比前一个数大，需要记录前一个数。

解二：

二．

1. 递归和分治

递归的一般思路：

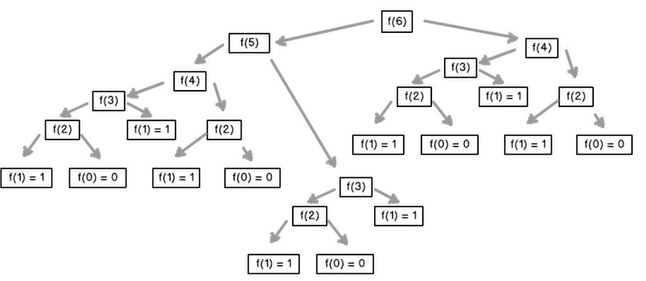
def f（level , p1,p2...） //注意level表示层级

if（）{} //函数内部先判断是否符合终止条件

Process //进行相应任务操作

f（level+1, p1,p2,..） //进入下一层 调用自己

Reverse //在完成下一层任务操作后的收尾工作



经常出现重复子问题 可以通过判重 或记录结果

分治 一般思路：

def divide\_conquer（problem，p1,p2,...） //把问题分小

if（）{} //判断 当前规模是否达到终止条件

Divide problem //把问题分成更小子问题

S1= divide\_conquer(problem[0], p1,p2,..) //递归地处理子问题

S2= divide\_conquer(problem[1], p1,p2,..)

S3= divide\_conquer(problem[2], p1,p2,..)

Result ={s1,s2,s3...} //把子问题合并

问题：

一：计算pow(x,n) 50

解一：循环n次 O(n)

解二：分治 把n一分为二 且两部分是相同的

X^n=x^ (n/2) \* x^ (n/2) n为奇数时与此类似 O（log2（n））

解三：非递归 即不自身调用

采用while（）不断把n减小 python实现采用了位运算

二：

1. 贪心

在对问题求解时，选择当前最优的解 适用范围有限

如 现有20 10 5 1 四种面额的纸币，要凑36元且总数最少。

每次选最大面额的纸币：20 + 10 + 5 +1

但当面额是 10 9 1三种时 要凑18元时：贪心会选择10+8x1而最好的是2x9

因为第一种情况 面额都是前面面额的倍数 贪心适用

适用场景：问题能够分解成子问题来求解 所有子问题的最优解能推出最终问题的最优解

子问题的最优解称为最优子结构

贪心算法与动态规划的区别：

贪心对每个子问题都选最优解（目光短浅），不能回退 动态规划则不一定选择子问题最优解而是保存多种运算结果，根据以前的结果对当前进行选择，能回退

问题：

一：买卖股票的最佳时机 122

给定一个数组表示一支股票连续几天的价格，只能持有一支股 每天可以买或卖，要求计算最多能赚多少钱

解一：dfs 根据是否持股 每天有两种选择进行dfs，在函数内设一个Max来保存最大 值 O(2^n)

解二：只考虑可能的赚钱路线 只要后面的价格比前面的价格高 就可以买入

暴力查找所有可能的买卖日期 算出能赚的钱 再选出最大值

此法的时间复杂度还不懂 O(n^n) ?

解三：贪心 计算每两天的价格看能否赚钱 因为6-1 =5-1 + 6-5

<https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-ii/comments/>

1. 剪枝：

1. 迭代与递归：

迭代是不断地循环如while，递归是自己调用自己

区别是是否有自身调用

1. 关于树，图问题的解决 大部分用到遍历
2. 使用关于指针的数据 要先判断是否为NULL
3. 数据注意初始化 不初始化将是随机数，若有判断大小就会出错