数据结构总结

- 数组
  - 类似python中的列表，列表中的元素类型可以不同
    - 示例
      - eg: 两数之和，给定一个整数数组和一个整数目标值，请你在该数组中找出和为目标值的那 两个整数，并返回它们的下标。
      - 输入：nums = [2,11,7,15], target = 9
输出：[0,2]
解释：因为 2+ 7== 9 ，返回 [0, 2] 。
      - 思路：使用字典，复杂度为O(1)

```
def twoSum(nums, target):
    #记录数据, eg: {2: 0, 11: 1}, 存储的是target减去每个num的差值和这个num的位置坐标
    hashtable = {}
    for i, num in enumerate(nums):
        #迭代
        if target – num in hashtable:
            # 如果这个差值在hashtable，说明这个数和hashtable中差值的坐标，hashtable[target – num]是返回差值的坐标
            return [hashtable[target – num], i]
        #如果不在hashtable，继续加入差值和这个num的索引坐标
        hashtable[nums[i]] = i
    # 默认返回为空，如果没有满足if中的2数之和的条件
    return []

res = twoSum(nums=[2,11,7,15],target=9)
print(f"2数之和为9的数的坐标是{res}")
```

- 链表
- 双指针
  - 一般出现在数组遍历中，一个从前往后，一个从后往前
    - 示例
      - 有序数组的平方
给定一个按非递减顺序排序的整数数组 A，返回每个数字的平方组成的新数组，要求也按非递减顺序排序
      - 示例 1:
输入：[–4,–1,0,3,10]
输出：[0,1,9,16,100]
      - 思路：因为给定的输入是非递减序列，最大值肯定在两头，逐个比对绝对值即可

```
def sortedSquares(A):
    # 从头开始计数
    i = 0
    # 从尾巴计数
    j = len(A) –1
    # 保存结果
    res = []
    # 从两头向中间靠拢
    while i<=j:   #注意这里用的i<=j
        if abs(A[i]) < A[j]:
            # 比较绝对值, A[j]**2是平方操作
            res.append(A[j]**2)
            #更改尾计数，向中心靠拢
            j –=1
        else:
            res.append(A[i]**2)
            i +=1
    return res[::–1]
res = sortedSquares(A=[–4,–1,0,3,10])
print(f"输入是[–4,–1,0,3,10]的平方后从小到大排列的结果是{res}")
```

- 字典
  - 复杂度为O(1)
    - 示例
      - 给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次
的元素。
      - 输入: [4,1,2,1,2]
输出: 4，4是只出现了一次
      - 思路：字典计数

```
def singleNumber(nums):
    # 用字典计数出现次数，找出只出现一次的数
    tmp = {}
    for num in nums:
        #遍历
        if num in tmp:
            # 如果出现过，那么计数就+1
            tmp[num] += 1
        else:
            # 第一次出现，那么添加一个，并且赋值为1，表示出现了1次
            tmp[num] = 1
    # 再次遍历字典
    for k, v in tmp.items():
        #遍历所有出现一次的，返回结果
        if v == 1:
            return k
print(f"给定列表[4,1,2,1,2]，其中只出现一次的数是{singleNumber(nums=[4,1,2,1,2])}")
```

- 堆栈
  - 堆栈和队列的底层都是数组，堆栈是先进后出，队列是先进先出FIFO
队列：排队，先到先得
堆栈：堆放木头，先放的在最下面，最后拿出来
用列表当队列和堆栈都是可以的，只需要pop选择索引的位置是第一个还是最后一个
    - 示例
      - 有效括号： 给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。
有效字符串需满足：
左括号必须用相同类型的右括号闭合。
左括号必须以正确的顺序闭合。
      - 输入：s = "()[]{}"　输入：s = "([)]"
输出：true　输出：false

```
def isValid(s):
    #初始化左括号的对应的有括号的字典，方便匹配
    dic = {'{': '}', '[': ']', '(': ')', '?': '?'}
    stack = ['?']  # 用列表当栈，用? 占位
    for c in s:
        #遍历输入的s
        if c in dic:
            #如果左括号，加入到列表中的尾部
            stack.append(c)
        #如果不是左括号，那么一定是右括号，那么从字典中找出左括号对应的有括号，判断是否和c相等
        elif dic[stack.pop()] != c:
            # 如果不相等，那么直接为FALSE，如果相等，循环下个括号
            return False
    # 判断是否还剩一个? 在列表中，如果是，那么说明ok，否则说明还剩一个左括号没匹配上，返回False
    return len(stack) == 1
s = "()[]{}"
print(f"输入{s}的判断结果是{isValid(s)}")
```

- 树
  - 二叉树遍历方法：
    - 先序遍历, 中序遍历, 后序遍历
      - 父在前就是先序，父在后就是后序，其他都是先左后右
        - 先序遍历: 先序遍历，父>左>右

```
class Solution:
    def preorderTraversal(self, root: TreeNode) –> List[int]:
        res = []  #结果列表
        stack = []  #辅助栈
        cur = root  #当前节点
        while stack or cur:
            while cur:  #一直遍历到最后一层
                res.append(cur.val)
                stack.append(cur)
                cur = cur.left
            top = stack.pop()  #此时该节点的左子树已经全部遍历完
            cur = top.right  #对右子树遍历
        return res
```

    - 复杂度O(lg(n))
- 回溯
  - backtracking 深度搜索算法，对比的就是广度搜索算法(又叫分支限界法), 难度+1
  - 回溯算法实际上一个类似枚举的搜索尝试过程，主要是在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。
  - 回溯法的代码套路是使用两个变量： res 和 path，res 表示最终的结果，path 保存已经走过的路径。如>果搜到一个状态满足题目要求，就把 path 放到 res 中
dfs算法就是深度搜索优先法，就是回溯法
    - 示例
      - 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。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]
]

```
def combinationSum(candidates, target):
    #存储最终的结果
    result = []
    def dfs(sumval, res):
        #表示target是非本地的，是从combinationSum来的， 示例中，这里target值是7
        # target从是从combinationSum来的原因是因为dfs(sumval, res)这个函数要进行递归查找，
        # target是不变的变量，所以从全局继承是最好的选择
        nonlocal target
        if sumval > target:  # 剪枝一:当前的总值大于目标值
            return
        if sumval == target:  # 当前值和目标值相等的时候,保存当前结果,并返回
            # res[:]是复制一个res列表的意思
            result.append(res[:])
            return
        for i in candidates:
            if res and res[–1] > i:  # 防止重复的方法是,不让其找在当前元素以前的元素
                continue
            #计算新的总和和新的可能列表，递归的查找
            dfs(sumval + i, res + [i])
    # 初始化第一个总和为0，列表为空的变量
    dfs(0, [])
    return result
print(f"给定序列是[2,3,6,7]，其中和为7的组合包括{combinationSum(candidates=[2,3,6,7], target=7)}")
```

- 迭代法
- 动态规划
  - 动态规划, 和分治法的区别如下, 小问题不独立
基本思想:将待求解的问题分解为若干个子问题，这些子问题之间不是互相独立的，但是如果在计算的时候>，可以保存子问题的答案，那么在需要重新计算的时候，可以直接从内容中获取计算答案，避免重复计算
    - 示例
      - 连续的子数组和
给定一个包含非负数的数组和一个目标整数 k，编写一个函数来判断该数组是否含有连续的子数组，其大小
至少为 2，总和为 k 的倍数，即总和为 n*k，其中 n 也是一个整数。
      - 输入：[23,2,6,4,7], k = 6
输出：True
解释：[23,2,6,4,7]是大小为 5 的子数组，并且和为 42。
      - 思考：利用字典记录每个位置的前缀和(包含当前位置)对k的取余结果, 要注意题目要求子数组长度必须不小于2

```
def checkSubarraySum(nums, k):
    #初始一个s记录数组的和，会除以k，是一个余数
    s = 0
    # 存储余数s和nums中每个数对应的坐标
    lookup = {0:–1}
    for idx, num in enumerate(nums):
        # 总和s加上当前数，更新总和s
        s += num
        if k != 0:
            # 总和除以k，得到的余数做为新的s
            s %= k
        if s in lookup:
            #子数组长度必须不小于2
            if idx > lookup[s] +1:
                return True
        else:
            #把余数s放入位置字典中
            lookup[s] =idx
    return False

res = checkSubarraySum([23,2,6,4,7],6)
print(res)
```

- 贪心算法
  - 贪心算法能做的动态规划都能做，但是动态规划更简单
- 分治法
  - 基本思想:将一个问题分解为多个规模比较小的子问题，这些子问题互相独立并与原问题解决方案相同，>
递归求解这些子问题，然后将这些子问题的解合并得到原问题的解。
    - 示例
      - 给定一个整数数组，找出总和最大的连续数列，并返回总和
      - 示例：
输入： [–2,1,–3,4,–1,2,1,–5,4]
输出： 6
解释： 连续子数组 [4,–1,2,1] 的和最大，为 6。

```
def maxSubArray0(nums):
    """
    O(n)复杂度
    :param nums:
    :return:
    """
    #记录当前的和
    cursum = 0
    #寻找最大的和，初始化列表中的第一个元素为最大的和
    maxsum = nums[0]
    for i in range(len(nums)):
        #变量整个列表，计算当前和
        cursum += nums[i]
        #和以前的最大和比较，找出目前的最大和
        maxsum = max(maxsum, cursum)
        if cursum <0:   #这一步要有
            # 如果当前和为负数，说明以前的值可以丢弃了，重新计算
            cursum = 0
    return maxsum
print(f"列表[–2,1,–3,4,–1,2,1,–5,4]的最大连续和是{maxSubArray0([–2,1,–3,4,–1,2,1,–5,4])}")
```