打印输出递归树

```
def printIndent(n):
    for i in range(n):
        print("   ", end='')
```

其通过空格表示层数，进而看出递归树层次，使用时，

- 在递归函数开头调用 `printIndent(count), count += 1`，并打印关键变量
- 在返回语句前调用 `count-=1, printIndent(count)`，并打印返回值

数组问题

分治法

- 二分法求和/搜索 O(log n)

双指针法：

- 快慢指针 O(n)，两数之和：找出在一个排序好的数组中满足两数之和的数，在最左和最右设置指针

```
slow, fast = 0, 0
while fast < len(nums):
    if ...:
        slow += 1
    fast += 1
```

- 多数之和问题：首先排序，而后求解
  - 三数之和 O(n^2)，固定首个变量，通过双指针解剩余的两数之和
  - 四数之和 O(n^3)，固定首个变量，解剩下的三数之和
  - N数之和 O(n^n-1)，递归求解NSum(n, nums, target)，固定首个变量，基于调用NSum(n-1, nums, target)获得答案。base case为n==2时即两数之和函数。
  - 注意去重，当该指针与上一指针指向的值相同时则continue

滑动窗口法：209 长度最小的子数组

- 思想：双指针法的一种，通过维护一个和 ≥ s 的长度最小的连续子数组，即动态窗口，以遍历所有满足条件的最小子数组
- 起始位置 j 移动：如果当前窗口的值大于s了，窗口就要向前移动（也就是该缩小了）。
- 结束位置 i 移动：为遍历数组的指针，是for循环里的索引。

```
for i in range(n):
    ...
    while j < i and ...:
        j += 1
```

- 字符串匹配Rabin Karp算法：找到匹配模式串p的索引，即遍历目标字符串，使用哈希值表示模式串，动态更新当前串的值，进而使匹配从O(NL)到O(N)。
- 连续子数组问题，尤其是带有限制的连续子数组问题，多使用滑动窗口解决。窗口右端以贪心结果，窗口左端以解除限制
- 维护滑动窗口内的最大值和最小值：使用单调队列，O(N)。例如对于最大值，若滑动窗口最右端的值比队列最末值大，则不断pop最末值。若滑动窗口最左端所对应索引超过队首的值对应的索引，则popleft。最后插入该值。
- 滑动窗口处理字符串问题，由于字符种类数为m，m多为常数即26，因此对于复杂问题，可以对每个字符分开用滑动窗口处理共常数m次，时间复杂度依旧为O(N)。
- 当求最大的问题时，滑动窗口可不收缩（即左侧每次只跟右侧移动一次），因滑动窗口的大小维护了所求。
- 长度为n的数组的子数组个数为1+2+3+...+n

二分查找：34 1870 1894

查找的对象，包括数组，或check函数（随x增大，函数结果增大），一定要呈单调性。

- 查找nums中首个大于等于target的数出现的位置。该模板保持答案位于二分区间内，二分结束对应的值恰好在答案所处的位置，或首个大于等于答案的值所对应的索引（即满足条件位置处右侧的值）

```
def binarySearch(target):
    l, r = 0, len(nums)
    while l < r:
        mid = (l+r)//2
        if nums[mid] >= target: // 可以换为check(mid)，因二分查找具有查高度等更多应用
            r = mid
        else:
            l = mid+1
    return l
```

返回满足条件位置处左侧的值：

```
def binary_search(self, nums, target):
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) // 2
        if nums[mid] >= target:
            r = mid - 1
        else:
            l = mid + 1
    return l
```

- 找两个排序数组合并后的第K个数字 4

  思想：利用二分查找，二分的对象是比较区间的长度。通过二分不断缩小当次比较区间长度K→K/2→...→1。当次比较区间为nums1[i, i+k//2]与nums2[j, j+k//2]，则根据大小比较决定，下一次比较区间为nums1[i, i+k//4]与nums2[j+k//4, j+k//2]，或为nums1[i+k//4, i+k//2]与nums2[j, j+k//4]。不断缩小比较区间直至1。

  该过程中，算法时刻保持了两个数组搜索总跨度（即索引0到当前比较区间末尾索引的距离）之和不变，即为K，进而保证了搜得的数字即为合并后数组的第K个数字。

- 寻找容纳K天的最小权重 1011 410

  思想：利用二分查找的思想，原题过于复杂时，且答案呈递增/递减的规律（如求最大/小值）则从答案反推，不断逼近范围。

前缀和问题 304

- 思想：O(n) 前缀和数组中存储的是原数组[:i]的和，通过前缀和数组，用于数组不会修改时，高效处理数组区间求和。list(accumulate(nums, initial=0))。前缀和数组向前多开一个。
- 拓展到二维：sums[i][j]表示(0,0), (i,j)矩形区域所围成区域之和，对(i,j), (i+h,j+w)矩形内数字求和=sums[i+h][j+w]-sums[i][j+w]-sums[i+h][j]+sums[i][j]

差分问题 370

- 思想：O(n) 差分数组中存储的是原数组两相邻元素间的差值，用于高效处理数组区间的修改。对于区间[l, r]的增加操作，将原数组转换为差分数组，执行以下步骤：1. 将diffArr[l]增加val，表示原始数组arr中从位置l开始的元素都增加了val。2. 将diffArr[r+1]减去val，表示原始数组arr中从位置r+1开始的元素都减去了val。然后，对差分数组求前缀和sums，sums[1:]即为修改后的原数组。差分数组向后多开一个。
- 拓展到二维：diffs[i][j]表示元素(i,j)与下一相邻元素的差值，对(i,j), (i+h,j+w)矩形内数字更新：diffs[i][j] += val, diffs[i+w][j] -= val, diffs[i][j+h] -= val, diffs[i+w][j+h] += val，而后对差分数组求前缀和sums即可。

求最小公约数和最小公倍数

- 思想：通过反复求模（欧几里得算法）求最小公约数，最小公倍数为两数之积除最小公约数
- ```
  def gcd(a, b): // 最小公约数
      while b:
          a, b = b, a % b
      return a

  def lcm(a, b): // 最小公倍数
      return abs(a * b) // gcd(a, b)
  ```

  辗转相除法：原理：gcd(a, b) = gcd(b, a % b)，即如果有一个数可以同时整除 a 和 b，那么这个数也可以整除 a, b, 和a-b的任意次。因此，它也可以整除 b 和 a % b（即 a - bq，q 为整数）

  n个数的最小公倍数：遍历，累积的结果与下一个数求最小公倍数，最后的结果即为n个数的最小公倍数。

随机数问题

- 思想：底层结构为数组和字典的应用，通过randint(a, b)产生随机数。

众数问题 169

- 思想：求众数（出现次数大于n//2）：摩尔投票算法。设置cnt与candidate，若cnt为0，则更新candidate为当前数值；若cnt不为0，若candidate为当前数值，则cnt += 1，否则cnt -= 1。遍历结束后，判断candidate出现次数是否大于n//2。
- 延伸：求出现次数大于n//3的数字：摩尔投票算法。设置两个cnt与candidate（因最多有两个该数字），若num为candidate1或candidate2，则cnt1或cnt2+=1。否则，将num更新到cnt1或cnt2为0对应的candidate，cnt1或cnt2设为1；若都不为0，则cnt1与cnt2都-=1。遍历结束后，判断candidate出现次数是否大于n//3。

  若求出现次数大于n//k，则设置k-1个cnt及其candidate
  ```

置换环 2459

- 思想：通过交换使0-n-1元素呈升序为置换环问题，遍历i，对于各未访问过且值不匹配位置的索引i寻找其所在的置换环（根据特点，一个元素只能位于一个置换环）：即令j=i，连续寻找对应位置上的值nums[j]，并加入vis，直至下一对应位置的值已在vis存在。该过程中涉及的元素构成同一个置换环。m个节点构成的置换环需要的使之成为升序的交换操作次数为m。

链表问题

需要时设置虚拟头结点，使代码更易实现与泛化。

可试图将链表转换为数组，以降低时间复杂度。

遍历法：

- 反转链表（也可递归实现）、链表相交

双指针法：

19 删除倒数第n个节点

- 思想：让快指针先走n步，当快指针为None时，慢指针指向的节点即是需要删除的。O(n)

142 环形链表 找到存在环形链表的起始处

- 思想：设置快慢指针，快指针每次走2步，慢指针每次走1步。若有环，他们一定会相遇。此后，经数学推导，在相遇处与head以相同速度同时出发，则其相遇地点即环形链表起始处。

LRU cache：

思想：基于collections.OrderedDict 即键值有序字典（是按照插入顺序的有序字典，也称linkedHashMap, 内部通过单独设置head与tail节点的双向链表实现）。使用某一键值对时，则通过.move_to_end(key) 函数将其移动到有序字典的最末，表示最近使用；cache满时，则通过.popitem(last=False) 函数将有序字典的第一个键值对删去。get与put操作均为O(1)

基于HashMap解决

1. 基于Set即dict()作为HashMap：

- 思想：通过in dict查找时，时间复杂度为O(1)，in list查找时，时间复杂度为O(N)。实际用defaultdict(int)实现，可在key不存在时不报错，便于添加元素，无需先判断是否key in hashmap.keys()。
    - 1 两数之和：用dict存还需要的值，遍历第二个数组查看该值是否存在。O(n+m)
    - 454 四数之和个数：用defaultdict存，遍历A与B数组，键为(A+B)后还需要的值，值为和为(A+B)的个数，而后遍历C与D数组，查询二者之和是否在键值中。O(n^2)

2. 基于数组作为HashMap：

- 思想：ord()获取ASCII码，用字符-ord("a")作为数组索引

基于数组的方法也能基于Set解决，但Map空间消耗更大，却更实用

collections.Counter：

- 思想：collections.Counter(字符串) 可直接统计每个单词（键）出现的次数（值）
- 1002 寻找共同字符：min(All_Counter) 后的非0结果即为共同字符，通过交集&实现该min
- 383 A是否能用B数组内元素组成：使用Counter后的A-Counter后的B，即为A中无法被B涵盖的值

list：

- 思想：in list操作在不能继续优化O的同时不能用dict的情况下使用

字符串

反转字符串：

- str.reverse() / reversed(str) / str[::-1]
- 双指针法：前后交换 O(N)
    - 'str'.join(list) 用于将list中的元素以str拼接
- 有些题通过整体反转，再局部反转

扩展字符串：

- 双指针法，先将原数组扩充到替换后的大小，再从数组末尾开始填充
    - 判断是字母 str.isalpha，是数字 str.isdigit()
    - str.split()，以空格/字符s对字符串切片，str.strip()，移除字符串头尾空格/字符s

KMP算法

- **字符串匹配 28：**
    - 思想：使用前缀表能够避免中途匹配失败时，模式串的回溯。其中，前缀表记录模式串str[0:i]中最长相等前后缀处的索引。
    - 定义：前缀：不包含最后一个字符的连续字符串，后缀：不包含第一个字符的连续字符串
    - 先算前缀表：通过定义两个指针 i 和 j ，假设 j 已经指向了该字符串前缀末尾位置（即-1），i 已经指向了该字符串后缀末尾位置（通过next数组实现，为前缀表-1，即初始化为-1，表示不存在相同的前缀）
        - 如果前缀末尾的下一位字符（j）与新后缀末尾（i）依然相同，则更新前缀末尾（+1），反之，则不断返回目前字符串（str[0:j]）前缀的前缀的前缀的...直至子串匹配上，或一直没匹配上则j回到-1（因为前缀与后缀相同，实则是寻找能匹配上原后缀的子串的原前缀的子串）
    - 再匹配：通过得到的最长相同前后缀数组，可以在失败时回溯到最近的模式串。
        - 实现时与前缀表算法相同，只是新增了返回条件，当前缀（str[0:j]）达到所匹配字符串末尾时，说明匹配成功，则返回。
    - python：s.find(subs)返回子串初始索引

- **寻找重复子串 459：**
    - 思想：使用KMP算法找出的最长公共前后缀的剩下的子字符串，即是重复单元。（可用得到的前后缀来证明）
    - 先用KMP算法算next表，则s[: len(s) - (next[len(s)-1]+1)] 则为重复单元，而后遍历依次判断即可

滚动哈希法 214 构造最短回文串（寻找最长回文前缀）

- 思想：利用滚动哈希，即对于字符串构造哈希值，O(N)遍历字符串，若left_to_right的hash值==right_to_left的hash值，则说明该字符串是回文串。left_to_right = (left_to_right * base + ord(s[i])) % mod；right_to_left = (right_to_left + power_base * ord(s[i])) % mod；power_base = power_base * base % mod；base, mod = 131, 10**9 + 7
- 应用：1316

前缀树法 336 寻找连接为回文串的组合

- 思想1：利用滚动哈希（记得mod），先计算出各word的hash与reverse_hash，以及各base。若hash(s1)*base[len(s2)] + hash(s2) == reverse_hash(s2)*base[len(s1)]+reverse_hash(s1)，则其(s1,s2)拼接为回文串
- 思想2：前缀树法，对words中的word依次slice各长度的前缀，以较长的字符串为基准，分情况讨论。当较长的字符串在前，如abcdece+dcba，则前缀为abcd, 后缀为ece，因此，判断前缀的reverse是否也在words中，同时后缀与其reverse是否相同；当较长的字符串在后，如dcba+eabcd，则前缀为e，后缀为abcd，因此，判断后缀的reverse是否在word中，同时前缀与其reverse相同。

二分长度法 1044 查找最长重复子串

- 思想：将子串的长度L作为二分的对象，若该长度L下找到，说明可能存在更长重复子串，则进行扩大L查找；反之，说明子串长度只能更小，则缩小L进行查找。

栈Stack和队列Queue

对于一些题，反向后，再用栈来匹配其模式。

用list实现栈：

利用list的append(), pop()。括号匹配 20， 相邻dup消除 1047

用deque实现队列：

利用python collections中的deque()，deque支持索引，self.deque[0]为最左端（left），self.deque[-1]为最右端（right），利用deque的append, appendleft, pop, popleft操作。

- **通过栈实现队列 232**
    - 思想：栈先入后出，队列先入先出。设置两个栈，一个输入栈，一个输出栈。输入栈pop的结果（栈顶，后入），push进输出栈（栈底，后出），以实现先入先出。
- **通过队列实现栈 225**
    - 思想：使用一个双向队列deque()实现（能从左右皆出入的队列）

- **通过堆实现优先级队列 求TopK元素 O(Nlogk)**
- 优先级队列即在插入和删除的过程中，其中元素呈一定大小关系自动排序。
- 思想：始终维护一个大小 <= K 的小顶堆。不断插入(freq, key)，最终堆内所剩的即为TopK。
  - 基于大顶堆/小顶堆（完全二叉树）实现，大顶堆：父节点 >= 子节点；小顶堆：父节点 <= 子节点。堆是队列结构。
  - 插入：插入为最后的子节点，然后从堆底到堆顶：
  - 弹出最大项：用最后的子节点替换父节点，然后从堆顶到堆底：

    交换不符合的子节点和父节点的值，重复直到堆的顺序恢复正常或到末尾 O(logN)
- 实现：根据collections.Counter统计的(key, freq)不断插入，堆大小超过K时则pop堆顶即最小值（及时删除为了减少内部查找时间，否则为O(NlogN)）。
  - python库中的heapq为小顶堆（实现大顶堆即以-num存入），通过heapq.heappush插入新值，heapq.heappop弹出堆顶值（最小/大值）heapq.heappush(heapname, (freq, key))。
  - 当堆中需比较的不仅为一个value，或涉及字符串时，则通过写一个Class的__lt__即比较方法实现。即heapq.heappush(pq, Pair(word, freq))，其中，Pair中的def __lt__(self, p): return self.freq < p.freq or (self.freq == p.freq and self.word > p.word
    - 当一个类作为lambda时，类无需self，只需__lt__方法，即__lt__(x,y)，如return x+y > y + x
- 底层实现：O(logK)。通过swim(上浮)与sink(下沉)和swap(交换)、parentL、parentR、children的方法，来实现insert与pop。上浮某个节点 A，只需要 A 和其父节点比较大小；下沉某个节点 A，比较 A 和两个子节点大小，若 A 不是最大/小的，则要把最大/小的子节点和 A 交换。
- 可通过底层修改，实现对堆中任意元素在O(logN)的删除，进而使用两个堆（最小堆存当前TopK值、最大堆存candidate），实现数组频繁插入/删除情况下的操作的TopK（也能通过内置SortedList实现，即平衡二叉树实现。因为堆的索引使之为平衡二叉树，因此可以模拟SortedList，即每次插入/删除为O(logK)）。

```
class MinHeap:
    def __init__(self):
        self.heap = []
        self.index_map = {}

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
        self.index_map[self.heap[i]] = i
        self.index_map[self.heap[j]] = j

    def _sift_up(self, index):
        while index > 0:
            parent = (index - 1) // 2
            if self.heap[parent] > self.heap[index]:
                self._swap(parent, index)
                index = parent
            else:
                break

    def _sift_down(self, index):
        left_child = 2 * index + 1
        right_child = 2 * index + 2
        smallest = index
```

```
        if left_child < len(self.heap) and self.heap[left_child] <
self.heap[smallest]:
            smallest = left_child
        if right_child < len(self.heap) and self.heap[right_child] <
self.heap[smallest]:
            smallest = right_child
        if smallest != index:
            self._swap(smallest, index)
            self._sift_down(smallest)

    def push(self, value):
        self.heap.append(value)
        self.index_map[value] = len(self.heap) - 1
        self._sift_up(len(self.heap) - 1)

    def pop(self):
        if not self.heap:
            return None
        root = self.heap[0]
        last_element = self.heap.pop()
        if self.heap:
            self.heap[0] = last_element
            self.index_map[last_element] = 0
            self._sift_down(0)
        del self.index_map[root]
        return root

    def remove(self, value):
        if value not in self.index_map:
            return

        index = self.index_map[value]
        last_element = self.heap.pop()

        if index < len(self.heap):
            self.heap[index] = last_element
            self.index_map[last_element] = index

            if last_element < value:
                self._sift_up(index)
            else:
                self._sift_down(index)

        del self.index_map[value]
```

- heapq.nlargest(k, nums, key=) 是根据key选取数组nums中前k大元素的python封装函数
- 堆的应用：合并K个有序链表
- **找大数流中的中位数 295**

- 思想：计算中位数O(1)，插入O(logN)。此题由通过堆来找第K大的数算法启发（即add后，首先对堆前几个元素pop出去，使堆里只留K个元素），专门设置另一个栈存这前几个元素，本题“前几个元素”即是数字个数的一半。算法使用两个堆，堆的大小之差不超过1。通过使存前一半数的堆由大只小，存后一半数的堆由大至大，两堆顶露出的数，恰为计算中位数所需的元素。

- **逆波兰表达式求值 150**
    - 思想：后缀表达式，无关运算符优先级与括号。
        - 通过使用一个栈记录其值，遇到数则进栈，遇到操作符则出栈两个操作数并运算后入栈。栈中最后一个数即为结果。
- **滑动窗口内最大值 239**
    - 思想：只维护窗口内的最大值，忽视最小值。
        - 通过队列实现（基于deque的新建class）一个从大到小的单调队列（和优先队列的实现有所区别）。该队列头部即该滑动窗口内的最大值。
    - 实现：将队头旧数pop（若队头数字没有被合并）后push新数。当新数大于队尾数字，则不断与其合并（即删去该数字），直至小于等于/队为空再插入该数字。

正则表达式运算 772

思想：基于栈实现，栈中存储计算结果与计算符，最后返回栈的sum即为运算结果：

- 对于数字，根据flag进行操作（flag初始化为1表示+）
- 对于+ -，直接将flag*num来append到栈中；对于* /，将stack.pop() * / num来append到栈中；
- 对于括号，将当前flag与左括号append到栈中，重设flag为1。匹配到右括号时则逐一pop直至左括号，并根据左括号前栈内的flag操作：
    - （同上）对于+ -，直接将flag*num来append到栈中；对于* /，将stack.pop() * / num来append到栈中。

单调栈

单调栈从栈底到栈顶可从大到小，也可从小到大。若要找两侧第一个最大，则为从大到小；反之，为从小到大.

求前缀和：accumulate(list, initial=0)

每日温度 739

- 思想：利用单调栈求解。O(n)。通过list实现，for元素，新元素大于栈顶元素时，while循环不断pop出栈顶元素，直至从栈底到栈顶呈由大至小。本题通过存储索引来求解

下一最大元素 496

- 思想：单调栈同上。作为单调栈的应用，每次返回指定索引的栈中值即可

下一最大元素（循环数组版本）503

- 思想：单调栈同上，由于循环，对两个原始数组拼接后的结果作为处理对象即可

接雨水 42

- 思想：以单个bar为计算单位，结果为∑min(leftMax, rightMax)-height[i]，是基于双指针的实现，逐步寻找leftMax与RightMax中较小的，因其为计算所依赖的。而基于单调栈的实现，是以多个bar为计算单位，设计栈底到栈顶呈由大至小的单调栈，雨水体积在每次栈压缩时进行增加，为(min(左高，右高)-底高) * 宽度。时间复杂度皆为O(N)

柱状图面积 84

- 思想：要找两侧第一个最小的，设计栈底到栈顶呈由小至大的单调栈。柱状图面积=max(前k个柱子的最矮高度*宽度k)

907 子数组的最小值之和

- 思想：动态规划O(N^2), 单调栈O(N)。即分别求最小值为arr[i]的子数组个数。最小值为arr[i]的子数组个数即到左边最近小于数的距离×到右边最近小于数的距离。因此即求arr中各数小于它的临近左右索引。因此即转换为通过单调栈进行求解。

2104 子数组范围和

- 思想：求子数组中最大值-最小值之和，转换为求最大值/最小值为k的子数组有多少个，加和求差即可。其中，最大值k所在的子数组个数 = (i - 前一个大于它的值索引) * (后一个大于它的索引 - i) * nums[i]；最小值k所在的子数组个数 = (i - 前一个小于它的值索引) * (后一个小于它的索引 - i)。即求四个数组，内容为该数组前/后第一个大于/小于它的值的索引。通过单调栈求解。max_r, max_l，min_r, min_l，其中max_l与min_l通过反转nums求解（最后需要用len(nums)-1-i，同时reverse()。该四个数组皆初始化为len(nums)，最后使得max_l与min_l边界为-1）。对于处理等于的情况，由于保证处理一次，不能多/少，因此设置同一方向有等于条件。例如max_r(>=), min_r(<=), max_l(>), min_l(<)

1856 子数组最小乘积的最大值

- 思想：求min_l与min_r，对应值即为nums[i]*sum([min_l+1, min_r-1])，同通过单调栈求解。其中，sum()通过前缀和进行求解，即从O(N^2)压缩到O(N).

321 n个个位数的序列中长度为k的最大子序列

- 思想：单调栈。遍历，若后方数大于前方，且剩余数字个数能填满栈，则抵消一个较小数，即贪心思想，将大数尽可能向前移动。stack中的前k位即为所求。

子数组和问题的解法：1.双指针 O(N) O(1) 2. 哈希表+前缀和 O(N) O(N)；3.前缀树+前缀和 O(NlogN) 4.单调队列O(N)

解决环形子数组问题，可分类讨论满足条件的是非circle的，还是circle的。

单调队列

跳跃问题（最大子序列问题）

思想：常用动态规划+单调队列求解（进而由O(NK)降低到O(N)），定义DP[i]为含有nums[i]的最大子序列和。

由于最大子序列数字可能不连续，无法施加贪心，同时也不能用数字值本身作为队列内的元素，需要施用过往子序列和即dp进行分析。

239 滑动窗口最大值

- 思想：求滑动窗口内最大值，用单调队列求解 O(N)。queue中存入队列时的索引，每次检查队首是否超出范围。对于每次入队，不断替换队尾，直至遇到队列中更大的。队头到队尾由大至小

1499 滑动窗口最大值，散点，不等式

- 思想：求滑动窗口内最大值 yi+yj+|xi-xj|，x是离散点，滑动窗口大小通过x_2-x_1 <= k定义。同样每次检查队首，而根据公式，保持队列为yi-xi > xj-xi的单调队列。每次首先对于队列更新maxV即y+y0+x-x0（根据当前规律，其为每次当前队列的最大值），最后返回maxV即可

862 和大于等于k的最短子数组

- 思想：在数组中记录索引，通过前缀求和数组（0-n+1，即sumV[i]为[0,i)），保证队列中sumV[i]-sumV[que[-1]] > 0，直至队列为空；当sumV[i]-sumV[que[0]] > k，则pop，并更新长度i-que[0]。

1425 邻距不超过k的最大子数组和

- 思想：维护que即k个缓存区，存放索引，dp[i]表示了从[0,k]内的最大子数组和。当que[0] < i-k 则说明该子数组和不适用，popleft出去。反之则从之前的dp中取最大或0，与当前num相加（即dp[i]=max(0, dp[j])+num (i-j <= k)。其中，最大通过单调队列append过程实现，即dp[i] > dp[que[-1]] 则 pop。

有序列表 SortedList

思想：通过 from sortedcontainers import SortedList; s = SortedList()。其是内部已维护排序好的列表。底层基于二分插入+动态数组实现。s.add(val), s.remove(val), s.pop(idx)

有序集合 SortedSet

思想：通过 from sortedcontainers import SortedSet; s = SortedSet() （或s=SortSet(key=lambda x:(-x[0], x[1]))）使用，操作包括s.add(x), s.remove(x)。其是内部已维护排序好的集合，常与二分查找结合使用。s.add(val), s.remove(val), s.pop(idx)的时间复杂度都是O(logK)（pop(-1)即从末尾弹出，为O(1)）。底层基于dict和SortedList，dict 存储每个元素及其在SortedList中的索引。

220 寻找滑动窗口内符合条件的数对

- 思想：在滑动窗口内维护有序集合。O(NlogK)

有序键值字典 SortedDict

思想：通过 from sortedcontainers import SortedDict; s = SortedDict() 。例题：588。在一个SortedDict中，键会按顺序存储在一个SortedDict中，而对应的值则会存储在一个普通的dict中。底层基于dict和SortedSet。SortedDict会使用SortedSet来保证键的顺序，并使用dict来保证值的快速查找和访问。

区间问题

思想：即线段问题，多为按照起点或终点排序后求解。结合DP/单调栈/区间和等，可通过该区间和将区间映射到x轴。

桶排序 1057

思想：当值在已知范围内分布时，使用桶排序，遍历范围内的所有值。例如，排序[0, 1000]内的数，遍历0到1000即可，当前位置存在数，则加入到已排序数列中，使排序的 O(NlogN) 优化到了O(N)。

树

[Important] 解题步骤：

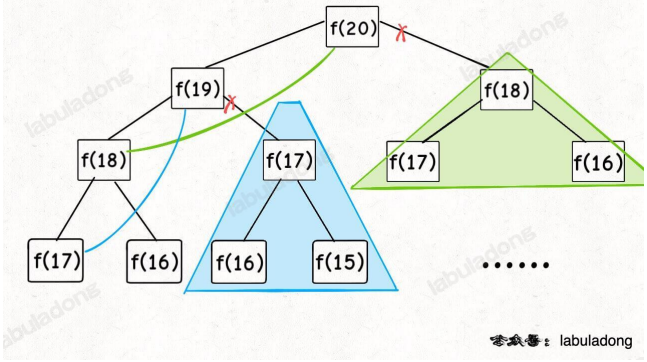
- **1. 看通过什么方式得出答案**：
  - 通过遍历得出答案：（即回溯）void traverse(...)，通过 更新外部变量 来计算结果
  - 通过分解问题得出答案：（即动态规划）int dp(...)，通过 子问题返回值 来计算结果
- **2. 看得出答案需要什么信息**
  - 需要上层信息：设置合适参数，通过参数传递
  - 需要下层信息：设置合适返回值，通过返回值传递

辅助：用字典dict存储节点、用字符串表示路径

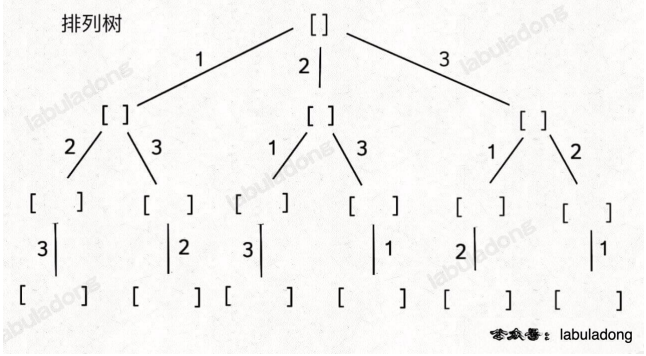
树的解决方式不应局限于以上traverse的解题方式，还可以将树转换为图、基于dict添加父指针等方式解决。

动归/DFS/回溯算法都可以看做二叉树问题的扩展，只是它们的关注点不同：

- **动态规划算法属于第二种分解问题的思路，它的关注点在整棵「子树」。**



- **回溯算法属于第一种遍历的思路，它的关注点在节点间的「树枝」。**



- **DFS 算法属于第一种遍历的思路，它的关注点在单一「节点」。**

二叉搜索树

- 和中序遍历配合使用。左子树上所有结点的值均小于它的根结点的值，右子树上所有结点的值均大于它的根结点的值

平衡二叉搜索树

- 左右两子树高度差不超过1

深度优先遍历（即DFS，基于堆栈Stack实现）

- 前序遍历：中左右；中序遍历：左中右；后序遍历：左右中
- 前序位置的代码只能从函数参数中获取父节点传递来的数据，而后序位置的代码还可以获取子节点通过函数返回值传递回来的数据。（**一旦发现题目和子树有关，就要给函数设置合理的定义和返回值，利用后序遍历。处理当前节点利用返回值，则后序遍历；若没有关系，则前序遍历**）
    - 递归法：三者皆相同的简单实现。
    - 迭代法：通过栈实现。前序遍历和后续遍历将各节点依次入栈/出栈。中序遍历只将左节点入栈，通过维护一个遍历的指针入栈左节点与处理中右节点。
    - 前序位置的代码在刚刚进入一个二叉树节点的时候执行；

      后序位置的代码在将要离开一个二叉树节点的时候执行；

      中序位置的代码在一个二叉树节点左子树都遍历完，即将开始遍历右子树的时候执行。

广度优先遍历（即BFS，基于队列Queue实现）

- 层次遍历（迭代法）102 515
    - 迭代法：while queue下每层有一个for循环

排序问题：912

- 快速排序：将数据分割成两部分，其中一部分数据比另外一部分都要小，再对这两部分数据快速排序，递归进行。时间复杂度O(NlogN)，空间复杂度O(logN)，最坏情况每次都选择出了最值，时间复杂度将达到O(N^2)

```
def quicksort(nums):
    if len(nums) <= 1:
        return nums
    pivot = nums[randint(0, len(nums)-1)]    # 随机选择基准元素
    left = [x for x in nums if x < pivot]    # 小于基准的元素放在左边
    middle = [x for x in nums if x == pivot] # 等于基准的元素放在中间
    right = [x for x in nums if x > pivot]   # 大于基准的元素放在右边
    return quicksort(left) + middle + quicksort(right)  # 递归排序左右两部分，合并结果
```

从代码可以看到，快速排序本质是二叉树的前序遍历：先找一个分界点 `p`，通过交换元素使得 `nums[lo..p-1]` 都小于等于 `nums[p]`，且 `nums[p+1..hi]` 都大于 `nums[p]`，最后对其左右子数组重复该操作。

应用：快速排序可用于寻找第k大元素 O(N)，即快速选择算法，相比通过最小堆实现O(NlogN)要快。因为快排本质上是每次选一个p，通过O(N)把小于其的元素放左边，大于其的元素放右边（类似二叉搜索树的构建，即最坏情况时变成链表），因此，若右侧元素个数大于k，说明元素在p右边，则在[p+1, r]中寻找；反之，说明在p左边，在[l, p-1]中寻找，若相等，则找到，类似二分查找。时间复杂度O(N)+O(N/2)+...+O(1) = O(N)。注意p需要随机设置，否则有可能退化到O(N)+O(N-1)+O(N-2)+...+O(1) = O(N^2)。

```
def quickSelect(nums, k):
    p = randint(0, len(nums)-1)
    left = [v for v in nums if v < nums[p]]
    middle = [v for v in nums if v == nums[p]]
    right = [v for v in nums if v > nums[p]]
    if len(right) >= k:
        return quickSelect(right, k)
    if len(nums) - len(left) < k:
        return quickSelect(left, k-len(right)-len(middle))
    return nums[p]
return quickSelect(nums, k)
```

- 归并排序：将数据不断折半，直到每个数据块只有一个元素，再按照拆分的顺序将每个数据块两两合并，合并的过程中进行排序。时间复杂度始终为O(NlogN)，空间复杂度O(logN)。

```
def sort(nums, lo, hi):
    if len(nums) <= 1:
        return nums
    mid = (lo + hi) // 2
    left = sort(nums, lo, mid)          # 排序左子数组nums[lo, mid]
    right = sort(nums, mid+1, hi)       # 排序右子数组nums[mid+1, hi]
    return merge(left, right)           # 双指针法合并左右两个有序数组
```

从代码可以看到，归并排序本质是二叉树的后序遍历：先对左右子数组排序，然后进行合并它们的操作操作。

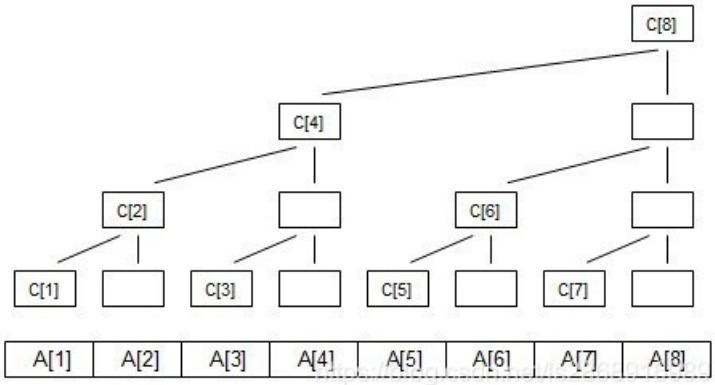
应用：归并排序可用于寻找reverse pairs，即小于x的数同时索引大于x的索引。其也可通过二叉索引数得出。前缀和范围可表达为reverse pairs，即可通过归并排序/二叉索引数求解

列表的归并排序：

- Up-Bottom：时间复杂度O(NlogN)，空间复杂度O(logN). 每次找mid，对mid及左边递归调用sorted，对mid右边递归调用sorted，同时每次合并两个有序链表。递归树深度O(logN)，因此空间复杂度O(logN)
- Bottom-Up：时间复杂度O(NlogN)，空间复杂度O(1). 由下至上，每次遍历链表长度，从2, 4, ..., 2**logN. 每次找mid，合并mid左与mid右，即合并其底层有序的两个链表。由于对于链表，只需记录一个节点，通过其next域，即可表示含N个节点，而数组需要记录N个节点。因此对于链表排序，空间复杂度可以做到O(1)。

- 冒泡排序：遍历数列，不断比较相邻数据，将小的交换到前面。重复n次，（或可只对未排序好的数列开展遍历，因第i次遍历可确定最终数列位置i上的元素）。时间复杂度O(N^2)。

树状数组（二叉索引树）



原理：二叉索引树的基本原理是使用位运算快速查询和修改数据。二叉索引树是一个数组(1-indexed)，数组的下标 i 和对应的元素值 v 表示了一个区间的信息。该区间由i的二进制最低位1的位置k确定，即区间 [i-k+1, i] 的值为v。因此更新某索引位置的值，则修改包含该索引的区间所表示的值；此后获得区间 [1, i] 值之和即前缀和，则可通过累加区间 [i-k+1, i] 的值在O(logN)内得到。

树状数组可高效解决区间内频繁更新的前缀和查询问题，是二进制的应用。

通过设置树状数组C，以O(logN)高效更新与查询了正常数组A。为使lowbit出现1，以索引1起始。

- 更新O(logN)：更改数组A时，修改包含数组A的所有树状数组C即可，包括本身x_k及其连续+lowbit(x_k)的树状数组值，直至末尾。如修改A[3]，则需要修改C[3], C[4]与C[8]。因有规则：C[4] = 3+lowbit(3); C[8]=4+lowbit(4)。lowbit(x)即二进制x中最低位1所对应的值，如lowbit(3)为011中最靠后的1，值为1；lowbit(4)为100中最靠后的1，值为4。修改时即加上更新的值delta。
- 查询O(logN)：查询前缀和。sum[7(111)] = C[7(111)]+C[6(110)]+C[4(100)]，即包括本身x_k及其连续-lowbit(x_k)的树状数组值，直至头部。

树状数组也可用于处理小于等于数x的个数（求小于等于时query的是idx_x，求小于时query的是idx_x-1）。即将值x作为二叉索引树的索引，小于等于该值的个数作为二叉索引树的值。（连续）

而处理和等于数x的个数（多数之和问题），即可利用哈希表的key直接计算。（离散）

```
class BinaryIndexedTree:
    def __init__(self, n):
        self.n = n
        self.c = [0] * (n + 1)

    @staticmethod
```

```
    def lowbit(x):
        return x & -x

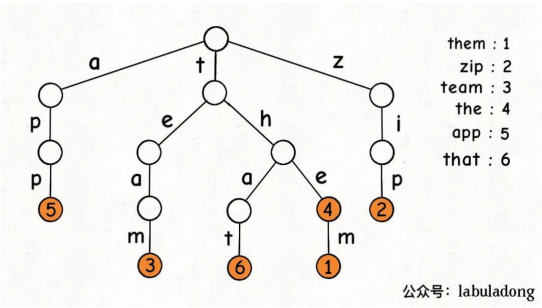
    def update(self, idx, delta): // delta为与原idx处值的差值
        while idx <= self.n:
            self.c[idx] += delta
            idx += BinaryIndexedTree.lowbit(idx)

    def query(self, x):
        s = 0
        while idx > 0:
            s += self.c[idx]
            idx -= BinaryIndexedTree.lowbit(idx)
        return s
```

二叉索引树的变体：求最长子序列（update与query中的从记录累加变为记录max，即小于x的数对应的最大值）

字典树（前缀树/Tire树）

前缀树即多叉树，用于能够高效处理前缀字符匹配等字符串操作。数据结构中存储了根节点TrieNode root。前缀树的边表示了各有效字符，树的节点为TrieNode，里面存储了children与is_word，表示到此节点为止是否是一个完整单词，节点中也可以存储值，表示该单词对应的属性，如个数。一个节点有 256 个子节点指针，但大多数时候都是空的，可以省略掉不画。结构如下：



前缀树常和前缀和搭配使用，即在节点中记录sum，其值为其子节点的所有值之和，即前缀和。前缀和在insert中更新，即对所有经过的节点加上所加节点的value。

匹配s与p的过程中，往往将更短的那个字符串构建前缀树。因为构建时时间复杂度是O(|str|)，搜索时也是O(|str|)

```
class TreeNode:
    def __init__(self):
        self.children = {}
        self.is_word = False

class Trie:
    def __init__(self):
        self.root = TreeNode()

    def insert(self, word: str) -> None: // 插入
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TreeNode()
            node = node.children[char]
        node.is_word = True
```

```
    def search(self, word: str) -> bool: // 搜索是否有字符串
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_word

    def startsWith(self, prefix: str) -> bool: // 搜索是否有以某前缀开头的字符串
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

    def erase(self, word: str) -> None: // 删除
        node = self.root
        def eraseHelper(idx, node):
            if idx == len(word):
                node.wordCnt -= 1
                return node.wordCnt == 0 and len(node.children) == 0
            c = word[idx]
            if c not in node.children:
                return False
            flag = eraseHelper(idx+1, node.children[c])
            if flag and node.children[c].wordCnt == 0:
                node.children.pop(c)
            return flag and len(node.children) == 0
        eraseHelper(0, node)
```

树序列化

- 思想：树的特异序列化。进而可通过实施字符串匹配O(N)来比较两个树。O(N)
- 实现：1. 以前序/后序遍历的顺序依次加入树的节点值，并在当前节点值之前加入^符号（即数字的分隔符，有时多位数字会出现数字歧义，如2 1 12与22 1 1的后序遍历皆为##1##122），None节点用#标记（即包含空指针信息）。

  ```
  def traverse(node):
      if not node:
          tree_str.append("#")
          return
      tree_str.append(("^"))
      tree_str.append(str(node.val))
      traverse(node.left)
      traverse(node.right)
  ```

  2. 以前序在左右子树开始和结束前各加上左括号与右括号。对于空右子树，其前后的括号可以省略；对于空左子树，若该根节点存在非空右子树，则空左子树前后的括号不能省略，否则可以省略。如root = [1,2,3,4]对应"1(2(4))(3)"；root = [1,2,3,null,4]对应"1(2()(4))(3)"。

- 如果序列化结果不包含空指针的信息，且只给出一种遍历顺序，则无法还原。如果有两种遍历顺序：

    - 如果是前序和中序，或者后序和中序，那么可以还原。
    - 如果是前序和后序，除非树中不包含值相同的节点，否则无法还原。
- 如果你的序列化结果包含空指针的信息，且只给出一种遍历顺序：
    - 如果是前序或者后序，那么可以还原。
    - 如果是中序，除非树中不包含值相同的节点，否则无法还原出。

树哈希化：

- 思想：数的特异哈希化。进而直接通过哈希值比较两个树。O(N)
- 实现：树哈希值 = ((左子树哈希值*左子树底数) % mod + (右子树哈希值*右子树底数) % mod + 根节点值) % mod

```
def traverseH(node, search):
    nonlocal basel, baser, mod
    if not node:
        return 1
    l = traverseH(node.left, search)
    r = traverseH(node.right, search)
    s = (l * basel % mod + r * baser % mod + node.val) % mod
    if not search:
        vis.add(s)
    return s
```

题

- **翻转二叉树 226**
    - 前序遍历递归/迭代法、后序遍历递归/迭代法、层次遍历可解决
- **对称二叉树 101**
    - 看成比较两个树，递归比较其中两个节点的左子节点与右子节点的值，以及右子节点与左子节点的值（满足对称结构）
- **相同二叉树 100**
    - 递归比较两个节点的左节点与左节点的值，以及右节点与右节点的值
- **深度/高度 104**
    - 深度：该节点到根节点的最长路径的边数
    - 高度：该节点到叶节点的最长路径的边数
    - 前序遍历：适合求深度，从根节点开始为0（树的最大深度就是其根节点的高度）
    - 后序遍历：适合求高度，从叶子节点开始为0
- **路径 257**
    - 将路径path存于迭代变量中，前序遍历，逐步延长path，将满足条件node对应的path append进out list中。
        - 涉及到回溯：给出下一递归时，变量前移(depth+1)；给出完毕后，复原该变量(depth-1)，而后继续其他递归。
            - 对于Primitive类型，可直接在函数参数中操作

        - 对于Reference类型的变量，需要在调用前后操作，否则需要.copy()保留现有变量
    - implementation时可能涉及全局变量，应将def写为内置函数，同时函数内声明 nonlocal 该变量名
- **完全二叉树节点个数 222**
    - 思想：完全二叉树的节点个数为2^depth-1。叶节点也是完全二叉树。
    - 实现：遍历每个节点，检验以该节点为根节点的树是否是完全二叉树（最左叶节点深度==最右叶节点深度），若是，则直接返回该树节点个数（2^depth-1），若不是，则返回该树节点个数（以左子节点为根的节点个数+以右子节点为根的节点个数+1）
- **平衡二叉树 110**
    - 思想：后序遍历获得并比较以左子节点为根的树的高度，与以右子节点为根的树的高度，相差大于1则非平衡二叉树
- **构造二叉树 106 617**
    - 根据遍历结果 106
    - 思想：对后序（前序）遍历结果从后往前（从前往后）遍历，对于每个order[i]，在中序寻找相等元素，将之作为pivot，以该节点为根节点对中序数组进行切割，后序/前序结果的切割根据中序数组大小确定。
    - 通过递归实现，返回该次递归生成的树的根节点，直至后序（前序）遍历数组为空。
    - 合并二叉树 617
    - 思想：都为前序遍历，每次递归返回根节点。
- **在二叉搜索树中寻找**
    - 思想：递归法/一次遍历法，对于每个节点，其右子树中所有节点的值皆大于该根节点值，左子树中所有节点的值皆小于该根节点值。无需回溯。
    - 二叉搜索树与堆的区别：堆是完全二叉树，是父节点值大于左孩子值，小于后孩子值（而非左子树的全部值，右子树的全部值）堆不一定是二叉搜索树，二叉搜索树因不一定是完全二叉树因此不一定是堆。
- **在二叉搜索树中插入**
    - 思想：遍历二叉搜索树，找到空节点插入元素即可
- **在二叉搜索树中删除**
    - 思想：遍历二叉搜索树，找到时，若该节点缺少右/左子树，则直接返回其左/右子树；若该节点具有双子树，则用左子树的最大节点/右子树的最小节点替换当前根节点；未找到时，即遍历到空节点，说明二叉树中没有该元素
- **二叉搜索树剪枝删除 669**
    - 思想：即删除所有不在范围内的节点。与删除不同，由于是二叉搜索树，剪枝可以直接把删除节点的左/右子树也连带剪掉。
    - 实现：后序遍历，每次返回该子树根节点
- **二叉搜索树累加 538**
    - 思想：反中序遍历（右中左），用sum记录所有之前遍历的累加值

- **构造平衡二叉搜索树 669**
    - 根据有序数组：二分法实现（以保证平衡），后序遍历，返回值为根节点（从数组构造搜索树答案不唯一）O(logn)
    - 根据链表：中序遍历顺序即为有序数组，因此使用中序先递归左得到左树，依次取list中元素作为node，node.left = left，再递归右得到右树。仍使用二分区间的left>=right作为返回条件，因此需要遍历获得链表长度 O(n)
- **验证二叉搜索树，在其中找最小差、众数**
    - 思想：由于中序遍历输出的是有序序列，判断中序输出是否单调递增即可。二叉搜索树中不能有重复元素。
    - 二叉搜索树中常用两个前后指针作比较：即在中序遍历过程过程中也可判断，在中序操作处设置一个全局pre指针记录上一节点值，与当前节点值比较即可。
- **回溯 236 235**
    - 若想自底向上对二叉树进行查找，则只能通过后序遍历（即回溯），实现从底向上的遍历方式，只有递归适用于回溯。
    - 后序遍历（左右中）就是天然的回溯过程，通过子树的返回值，来处理中节点的逻辑。
    - 有返回值的都是后续遍历，回溯有返回值，需将所有节点都遍历一次，后续遍历允许特殊情况的前置判断（在调用左右子树之前），但非特殊情况判断放于调用左右子树之后
    - 查找二叉搜索树则无需回溯
- 涉及到二叉树的构造，一定前序，都先构造中节点。
- 求二叉树的属性，
    - 普通二叉树：一般是后序，因为要通过递归函数的返回值。
    - 二叉搜索树：一般是中序，因为根据有序性。
- **两节点最近公共祖先 1650**
    - O(N) O(1)：求二者到最远祖先的距离之差，而后让长的先走该距离之差步。则二者将在第k步同时到达公共祖先
    - O(N) O(N)：用set记录一个节点到最远祖先路径上的节点，看另一个节点遍历过程中是否在该set中

回溯

解决N叉树状结构问题。因此，回溯和树的解题思路一致，通过边构造回溯树，若回溯树信息只需由上至下传递（函数名多叫做backTracking），则无需定义返回值和记忆状态。若回溯树信息取决于下层（函数名多叫做dfs），则需定义返回值，并考虑记忆状态。

本质为暴力查找。通过纵向遍历中的横向遍历完成对整个树的查找。

时间复杂度：O(b^d)，d为树深度，b为每个分支的候选个数。分析时，画树形结构的每一状态。

回溯搜索也称为dfs搜索，在数组中执行的时间复杂度为O(2^N)。

通过递归自顶向下解决问题包括：

1. 回溯：传递的参数大多从头，向后向传递，表示的是树的纵向层数
2. 分治：传递的参数是中间，向两侧传递，表示的处理的对象即树的横向

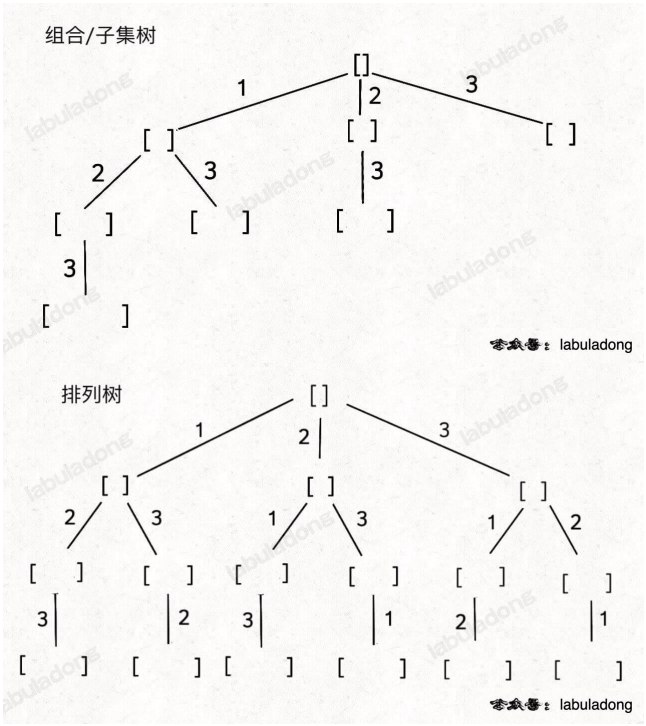
实际时间复杂度即暴力法的时间复杂度。

```
void backtracking(参数) {
    if (终止条件) {
        存放结果;
        return;
    }

    for (选择：本层集合中元素（树中节点孩子的数量就是集合的大小）) {
        处理节点;
        backtracking(路径，选择列表); // 递归
        回溯，撤销处理结果
    }
}
```

组合 （组合问题、切割问题、子集问题）

- 思想：for循环（当前元素索引~末尾）不断添加元素，返回条件：当前数组长度为所需长度。回溯：递归前添加for循环当前元素进tmp，递归之后pop其，继续for循环。
- 实现：当len(tmp) == k时，将tmp添加进out中，tmp为全局变量。基本在递归中维护startIndex。
- 优化：即剪枝，当剩余元素不够填充数组时则return。往往先对数组排序（以在for循环中可以直接return而非continue）
- 去重：对同一树层去重。比如组合之和==target的组合不允许出现重复数时，是同一树层去重，即不对该层中第二次出现的数继续展开递归，即nums[i] != nums[i-1]\ (nums有序时) 或 used = set(), nums[i] not in used
- 注意：每次递归的下一层是从i+1开始的，不应写成startIndex+1
- 组合 77
- 回文串切割 131
    - 思想：组合 + 动态规划
- 子集 78 90
    - 思想：仅收集结果处的区别。子集问题要收集所有节点的结果，而组合问题是收集叶子节点的结果。因此在每次递归开始时，将tmp加入out。组合是在最终满足条件时将tmp加入out
    - 去重：若子集有重复元素，则去重思路同组合
    - tmp[:]可替代tmp.copy()
- 时间复杂度：O(2^n)，因每一个元素的状态为取与不取。(若结果填入返回数组，需要额外O(n))



排列（排列问题、棋盘问题）

- 思想：对同一树枝去重，树的纵向不能存在相同元素。树深度相同。
- 全排列 46 47
  - 实现：使用used数组标记该path是否使用过元素，若未使用过，则加入。
  - 时间复杂度：O(n!)，也可从树形图分析得出
- 旅程选择 332
  - 思想：基本排序思想，通过defaultdict存旅程，内部排序，used去重。
  - 返回时：若存在满足的旅程即返回（通过设置递归函数返回值，成功则为True，以及时取消下一步递归）
- N皇后 51
  - 思想：基本排序思想，used去重，仅额外多了游戏的规则作为continue条件，即若符合规则（不在同列、排、对角线（通过 len(tmp)-j == abs(i-tmp[j]) 判断对角线），则继续递归
  - O(n!)
- 数独 37
  - 思想：排序思想，二维遍历，通过返回True与False及时返回找到的已知解，额外数独规则的check continue
  - O(9^空缺数)

总结来看，去重方法有三种：

1. 设置startIndex，遍历时在(i, len(nums))间遍历，为组合所用，因startIndex会导致组合提前结束
2. 设置used数组，不允许每层（横）出现相同的元素
3. 设置used数组，不允许每个path（竖）出现相同的元素
   - 注意当nums中存在重复元素时，横向的去重与竖向去重不应该冲突，即横向去重额外增加条件：元素nums[i-1]在该层未被纵向去重，即used[i-1]为0
     - 也可使用set替代used，但used更快

贪心算法

贪心有时候就是常识性的推导，若可以局部最优推出整体最优，且想不到反例，那么就试一试贪心。

贪心算法解题步骤一般分为四步，没有像二叉树和回溯算法固定的模板：

- 将问题分解为若干个子问题
- 找出适合的贪心策略
- 求解每一个子问题的最优解
- 将局部最优解堆叠成全局最优解

局部最优→整体最优

局部最优的定义有两种：

1. 对于求值：局部求得最值
2. 对于求满足条件：局部已满足题意

- 分发饼干 455 O(logn)
    - 思想：对饼干和胃口排序，将最小的饼干分给最小的胃口。
    - 实现：while循环胃口，找到第一个满足的饼干，直至胃口/饼干为空。
- 摆动序列 376 O(n)
    - 思想：局部：试图产生最多的局部峰值，单调坡度上的节点，最多产生两个局部峰值。整体：对于一个单调序列，无论如何节点个数多少/与后续如何连接，其有效峰值个数=2，其他皆为无用。因此保留每个单调序列的两个节点，跳过无用的节点，即可达到最长的摆动序列
    - 实现：遍历，对单调性特异(相等时算作一个单调序列)的连续两个节点计数，反之跳过。
- 最大子数组 53 O(n)
    - 思想：从后面看贪心：负数加上一个和永远会更小，因此要永远保证当前子数组和不拖累后续总和（即和>0），以越加越大。
    - 实现：遍历，记录最大子数组和，若当前子数组和<0，则及时断掉序列。比使用动态规划更优
    - 环形最大子数组 918 O(N)
        - 思想：分类讨论：有环/无环。有环的最大子数组和=sum(nums)-最小子数组；无环的最大子数组和=最大子数组。环形最大子数组和=max(有环的最大子数组和，无环的最大子数组和)。最大子数组和最小子数组利用滑动窗口+贪心求解。
- 买卖股票 122 O(n)
    - 思想：股票利润可以分解，即 3 - 0 = 3 - 2 + 2 -1 + 1 - 0。因此即使未提取卖出，此次利润也可由之前每天的利润想加而得。局部收集所有每天的正利润，全局则可得最大利润。
    - 实现：遍历，记录所有正的相邻和，即为总最大利润。
- 跳跃 45 55 O(n)
    - 思想：局部贪心要跳最大步，即关注可跳跃范围即可，不局限于跳的具体步数。
    - 实现：遍历，在当前可跳跃范围内，查找下一可跳跃范围最值，看是否能跳到末尾。跳跃范围的扩展轮数，即为最少达到末尾所需的步数。
    - 可转换为区间问题求解
- 数组取反求和 1005
    - 思想：对负数每次取具有最大绝对值的反，若没负数，则对正数每次取最小绝对值的反

    - 实现：根据绝对值由大到小排序，若当前为正，则跳过，为负，则反转。若没有负数存在还剩余反转次数，则持续反转最后一个正数（即最小正数）
        - 排序：A = sorted(A, key=abs, reverse=True)，或A.sort(key=abs, reverse=True)
        - A.sort(key=lambda x: (-x[0], x[1]))：根据x0从大到小排序，当x0 (维度1) 相同时，再根据x1 (维度2) 从小到大排序
- 油箱剩余 134
    - 思想：即要求长度为n的最大连续子序列问题，后面看贪心，sum大于0才与后面newVal相加，保证始终局部最优
    - 实现：gas-cost求rest数组，一旦子序列长度为达到n且之和小于0，则需重新设置序列起点为不满足条件处的下一点（因若[i, k]的sum<0，根据连续子序列的局部最优思想，[i+1, k]也无法满足），直至由该起点可扩展至长度为n，或起点不在[0, n-1]内，则遍历结束说明未找到。
- 糖果分配，大的要分大值，最少为1 135
    - 思想：局部最优：拐点处（最底）的人只分1个，对于具有更大权重的人仅多分1个。
    - 实现：通过两个遍历实现，（即替代了先求各拐点，再跟据左右依次逼近顶端值的操作，因通过Max操作实现效果相同），首先从左遍历，若i+1 > i值，则i+1值为i值+1，反之为1；再从右遍历，若i-1>i值，则i-1值为i值+1，反之为1。各索引最终值则为得到的两个遍历数组值的Max。
- 重建身高队列 406:
    - 思想：涉及两个维度时，先固定一个合适的维度，再确定另一个。局部最优：基于根据排序后（身高高的在前，k大的在后）的[h, k]不断进行插入，插入结果对已插入的数无影响；且由于前面插入的数一定大于等于该数，因此直接插到第k位即可。逐步插入扩展该数组，达到全局最优。排序O(nlogn)，插入O(n^2)
    - 实现：通过A.sort(key=lambda x:(-x[0], x[1]))实现排序；list插入有insert(index, val)方法
- 重叠区间问题（下四题）
    - 都先排序，尽可能让相邻区间叠在一起，这样后续不会对前造成干扰，保证最少的操作
    - 区间问题的贪心逻辑相同，仅后续针对题意的处理不同
- 重叠区间（射气球） 452 O(nlogn)
    - 思想：找涵盖数组个数最多的重叠空间。进而发射消耗弓箭数量最少。
    - 实现：基于数组左边界进行排序，不断维护重叠空间右边界，若当前数组与右边界具有重叠（即数组左边界小于等于右边界，则根据当前左边界维护右边界）；反之，则需再射一发，重置重叠空间（右边界设置为重叠空间中首个数组的右边界）
- 非重叠区间 435 O(nlogn)
    - 思想：找涵盖数组个数最少的重叠空间，即非重叠区间，通过排序后，不断去除右边界大的那个区间，最大程度防止后续重叠
    - 实现：基于左边界进行排序，维护区间右边界。若i+1的左边界<i的右边界，则重叠，除去。
- 划分字母区间 763 O(n)
    - 思想：找非重叠区间。维护最右边界，在当前到最右边界中对其不断更新，若抵达最优边界，则当前即为划分点。
    - 对于贪心算法，可视为非重叠区间问题看待，将字母[首次,末次]的出现索引作为区间边界。排序后同前题去除重叠区间，剩下非重叠区间的边界即为解。
- 合并重叠区间 56 O(n)
    - 思想：找最广区间。若前后重叠，则合并，最右边界为Max(i-1,i)；反之重置区间为当前。

- 单调递增数字 738 O(n)
    - 思想：不符合的元素直接用9占领。从低位往高位遍历，一但高位小于低位，则将该位及之后数字全赋为9，高位-1，依次类推。
- 覆盖二叉树 968 O(n)
    - 思想：贪心：由于叶节点数目在二叉树中最多，因此对树从后向前判断，不将摄像头放在叶节点，以使用最少的摄像头。根据左右子节点的覆盖情况（存在无覆盖时）插入摄像头
    - 实现：后序遍历，返回值为该节点覆盖情况（有摄像头/有覆盖/无覆盖）
- 区间内数最小差 910 O(nlogn)
    - 思想：区间内每个数需要+k或-k，求操作后的最小的极差。由于对于任意两数i与j，假设i < j，则为获得其极差，要么同时+k，要么同时-k，要么i+k且j-k，即根据贪心策略，以上结果均小于i-k且j+k的极差。由于对于任意两个数均满足上述条件，因此相邻两数也满足如上条件，即一定存在一个拐点，在其之前所有数均选择+k，在其之后所有数均选择-k，以保证作为最小极差的candidate。基于此pattern，遍历全部的点依次作为拐点，以获得最小值。

动态规划

动态规划一般脱离了递归，自底向上进行递推。递归一般是自顶向下。

DP分为两种：Up-Bottom（Recursion+Memorization，记忆空间非指数时可以使用，否则为Brute Force）；Bottom-Up（Iteration）

动态规划大多和树的第二种解法（分解解法）相似，只是树是通过递归先进行子问题分解而后合并，而动态规划通过从小到大遍历直接合并子问题（即Bottom-Up较多）。

- 动态规划是基于上一状态推导
- 而贪心没有状态推导，是从局部直接选最优

实施DP的条件要有最优子结构，即没有重叠子问题。如果有，则转换成没有，则可以实施DP。

复杂的求最值问题大多使用DP，图论的最短路径最值问题除了可以用djstra，还可以实施DP。

如果动态规划不能解，即限制条件不够，即DP定义导致状态转移有误，此时应及时更换DP定义来求解，如逆向DP等。DP也可通过记忆化搜索（非记忆化时即为暴力回溯）求解，以提供直接的思路。

复杂问题DP根据1. 状态 / 2. 空间大小 来定义，确定状态转移的推导则可解。

一些动态规划可以前置地通过人为策略先进行处理，简化递推公式和空间。

步骤：

1. 确定dp数组及下标含义
2. 确定递推公式
3. dp数组如何初始化
4. 确定遍历顺序
5. 举例推导dp数组

DP定义：

- 题干所求（大多数）
- 若难以直接获取全局所求，则间接给出局部，而后累加得到

转移公式可从以下思路推导：

- 对象序列 (如回文串) 的特征
- 状态转移图
- 二维表

基础题目

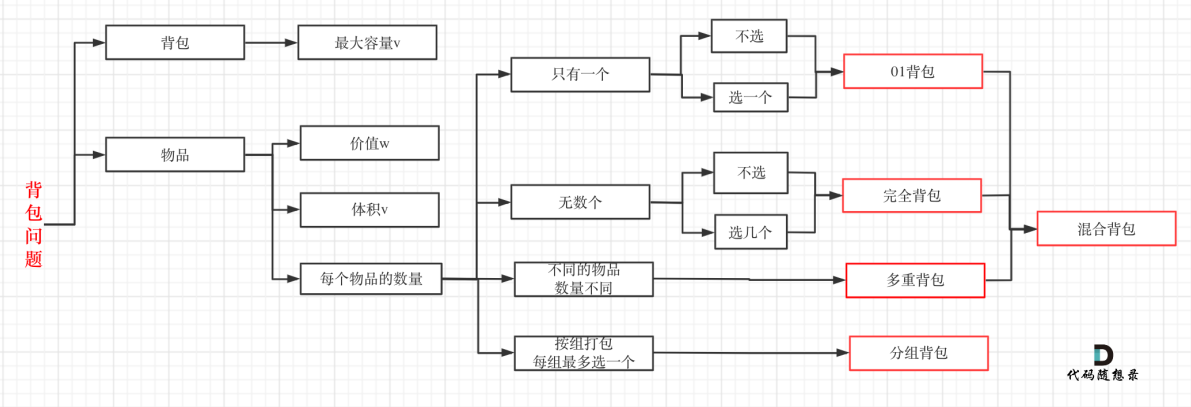
- 斐波那契 509 O(n)
  - 思想：DP定义为斐波那契数列。DP[n] = DP[n-1] + DP[n-2]；DP[0] = 0, DP[1] = 1。动态规划的递推解法为O(N)，而递归的解法为O(2^N)
- 爬楼梯 70 O(n)
  - 思想：DP定义为爬到第n阶台阶所需的步数，由于一次走1步/2步，因此DP[n] = DP[n-1] + DP[n-2]。DP[1] = 1，DP[2] = 2
- 最小爬楼梯cost 746 O(n)
  - 思想：DP定义为爬到第n阶台阶所需的最小cost，DP[n] = min(DP[n-1], DP[n-2]) + cost[n]，最后返回min(DP[n-1], DP[n-2])，DP[0]=cost[0], DP[1]=cost[1]（假设第一步花费，最后一步不花费）
- 不同路径 62 63 O(mn)
  - 思想：DP定义为到(m, n)需要的步数，DP[m,n] = DP[m-1, n] + DP[m, n-1]。初始化时额外增加外圈，值为0，DP[1,1]=1。若存在障碍，将存在障碍的DP设为0即可。

拆分推导

当dp比较复杂时，依次遍历所有可能的拆分，以获得最终结果

- 数拆分最大乘积 343 O(n^2)
  - 思想：DP定义为数n拆分后的最大乘积：由对其再次拆分后所有可能的乘积推导，包含 j*(i-j)（拆分i为两个因子的乘积）与 j*DP[i-j]（拆分i为三个以上因子的乘积）（i由前往后逐步推导，j 从1到i-1）。DP[i] = max(j * (i-j), j * DP[i-j])，对每个j不断更新DP[i]的最大值。初始化DP[2] = 1。
- 二叉搜索树的个数 96 O(n^2)
  - 思想：DP定义为n个元素组成的二叉搜索树的个数，由二叉搜索树的组成推导（忽略具体元素是什么，只观察树形态），发现是由包含j个元素的左搜索树+根节点+包含i-j-1个元素的右搜索树组成。因此DP[i] = DP[j] * DP[i-j-1] (0<=j<=i-1)。包含0个节点的搜索树只有一种形态，因此初始化DP[0]=1。

背包问题 (*Knapsack Problem*)



```
for i in 物品 (外层物品：组合数；外层背包：排列数):
    for j in 背包 (大→小：01背包；小→大：完全背包):
        dp[j] = max(dp[j], dp[j-w_i]+v_i)
```

- **01背包 O(MN) M为物品数，N为背包容积 物品最多装一次**
    - 二维思想：DP[i][j]：将0~i的物品放入容量为j的背包所得的最大价值。由是否放物品i推出：DP[i][j] = max(DP[i-1][j], DP[i-1][j-w_i]+v_i) 。初始化DP[0][k] = 0或w_0, DP[k][0] = 0
    - 一维思想：DP[j]：容量为j的背包所得的最大价值。DP[j] = max(DP[j], DP[j-w_i]+v_i)。相当于把dp[i][j]中i的维度去掉。初始化为0即可
    - 注意：二维dp正序遍历，即背包容量从小到大；一维dp倒序遍历，即背包容量从大到小，这是为保证物品i只被放1次，从后往前循环，基于推导的前方数组依然是老数组，即可以防止物品被拿两次（二维是由dp[i - 1][j]计算而来，本层dp[i][j]并不会被覆盖）。
    - 注意：一定要先遍历物品，再遍历背包容量。因为背包容量需要倒序遍历，反之的话背包里只可能会被放入一个物品

    求最大值/能否装满

    划分相等子数组 416

    - 思想：判断划分等问题时，又要比较各划分后的总值，方式利用动态规划。将本身的数值视作背包问题中的值，建立一个容积为（各子集）总值的背包，当做背包问题来解。

      DP[j]：容量为j的背包可装的最大值。重量和价值皆视为nums[i]。DP[j] = max(DP[j], DP[j-nums_i]+nums_i（nums_i<=i<=sum(nums/2))。所有元素（物品）都遍历完后，看最后所得的DP，即若当DP[sum(nums)/2] = sum(nums)/2时，说明存在相等子数组。（若sum为奇数，则直接不存在）

    最后一块石头重量 1049

    - 思想：同上，根据分析可知，最后一块重量为sum-2*(x_i+x_j+x_k+x_...)，因此求出与sum(stones)//2 最接近的石头重量之和即可。利用同上题的DP[i]定义与推导即可实现。这里注意target要小于等于sum/2（因最后一块重量不能为负），所以target=sum//2，即下取整

    求个数

    目标和 494

    - 思想：最大值问题是求背包中能装的最大的值，个数问题是求装满该背包的种数。DP[j] += DP[j-nums[i]]。初始值为0，其中DP[0]=1。思想不变，依旧为O(MN)
    - 该组合问题也可用回溯，但是为O(N^M)的暴力复杂度

    求长度

    一和零组合 474

    - 思想：求满足条件的最大长度问题，即DP[j][k] = max(DP[j][k], DP[j-nums[i][0]][j-nums[i][1]]+1)，初始化为0
- **完全背包 O(MN) 物品装的数目不限**
    - 思想：状态转移等思想皆相同。只是遍历时，内层的背包容量从小到大遍历，（0-1背包是从大到小，以防物品装多次，使前序皆为旧状态）。同时，判断是求排序/组合数。
    - 对于完全背包问题，DP[j]是基于前序新状态推导，而对于0-1背包，DP[j]是基于前序旧状态推导。
    - 与0-1背包不同，遍历物品/背包的内外层循环顺序不同：

1.求排序数，或顺序敏感的题目：必须背包在外，物品在内

2.求组合数：必须物品在外，背包在内

3.求最小数：二者皆可

- 先遍历物品，再遍历背包，求出的是组合数{a,b}，因为物品加入背包有先后顺序，先一定在后的前面；先遍历背包，再遍历物品，求出的是排列数{a,b}, {b,a}，因为背包容量中每一个值都经过了各物品的计算

零钱兑换 518

- 思想：硬币数量不限，求组合数的完全背包问题，求个数问题，DP[j] += DP[j-nums[i]]，初始化为0，DP[0]=1

组合之和 377

- 思想：求排序数的完全背包，求个数问题。若题目要去列出，则需要使用回溯（暴力算法）

爬楼梯 70

- 思想：求排序数的完全背包，求个数问题

零钱兑换 322

- 思想：求最小数的完全背包，DP[j] = min(DP[j], DP[j-nums[i]]+1)，需要注意的是，求最小值，则初始化DP成一个很大的数。本题DP[0]=0

完全平方数 279

- 思想：求最小数的完全背包，把完全平方数看成物品，DP[j] = min(DP[j], DP[j-i*i]+1)
- 对于最小数，遍历顺序皆可，因为排序数和组合数的最小个数相同

单词拆分 139

- 思想：求排列的完全背包，即使不是求值，由于单词有序，是顺序敏感，因此必须用排列求法。将DP[j]定义为长度为j的字符串s可以由字典中的单词构成。当word匹配上且DP[j-len(word)] == True时，DP[j] = True。初始化DP[0]=True
- 对于不同容量的背包，去遍历物品，这样物品才能有顺序地先后使用多次。

打家劫舍系列 198 213 337

- 线型，相邻的房子不能打劫。思想：DP为到第i家时打劫的最大钱数，根据规则，若选了nums[i]，则nums[i-1]不能选，则加前面的和DP[i-2]；若不选nums[i]，则加前面的和DP[i-1]。则DP[i] = max(DP[i-2]+nums[i], DP[i-1])
- 圈型，相邻的房子不能打劫。思想：分类讨论。DP定义同上。分成不偷第0间房/第len(nums)-1间房的情况讨论。实现时，DP计算方式相同，前者传入DP[1:]计算，后者传入DP[:len(nums)-1]计算。最后DP的max即为结果。
- 树型，相邻的房子不能打劫。思想：分类讨论，后序遍历。对当前节点分为偷/不偷，偷的val = node.val + l[0] + r[0]，不偷的val = max(l[0], l[1]) + max(r[0], r[1])，val1与val2作为返回值，最后返回max(root的val1, root的val2)。或者在实现时，直接返回当前DP, 左孩子DP值，右孩子DP值。返回的当前DP值=max(node.val+左孩子的孩子的DP值+右孩子的孩子的DP值，左孩子的DP值+右孩子的DP值)

买卖股票系列 121 122 123 188 309 714（状态转移）

- 只可买卖一次。思想：根据当前状态对DP分类，有两个状态：未持有股票时(未买/已卖)的最大利润DP[i][1]；持有股票时(已买/未卖)的最大利润DP[i][0]。最后DP[-1][1]即为所求。推导：DP[i][1] = max(DP[i-1][0]+prices[i], DP[i-1][1])（即要么卖掉，要么维持），DP[i][0] = max(DP[i-1][0], 0-prices[i])（即要么新买，此时是未买状态，因只可买卖一次因此利润为0，要么不卖）

- 可买卖多次。思想：同上，有两个状态：持有/不持有。只是对于持有股票的推导不同：DP[i][0] = max(DP[i-1][0], DP[i-1][1]-prices[i])，即由于可再次买，因此状态转移时需加入上回利润DP[i-1][1]（因前一题状态转移不能回转）
- 只可买卖两次。思想：动态规划本质为状态转移。本题有四个状态。分别为买一次、卖一次、买两次、卖两次。最后卖两次对应的DP即为所求。DP[i][0] = max(DP[i-1][0], -prices[i]), DP[i][1] = max(DP[i-1][1], DP[i][0]+prices[i]), DP[i][2] = max(DP[i-1][2], DP[i][1]-prices[i]), DP[i][3] = max(DP[i-1][3], DP[i][2]+prices[i])。初始化：DP[i][0]=-prices[0] , DP[i][1]=0 , DP[i][2]=-prices[0]，DP[i][3]=0（可理解为，第一次买入：当天买入；第一次卖出：当天买入后当天卖出因此为0；第二次买入：买、卖、买；第二次卖出：同理，为0）
- 只可买卖K次。思想：同理，定义2*k个状态，分别为买一次，卖一次，...，买k次，卖k次。DP[i][j]=max(DP[i-1][j], DP[i-1][j-1] +/-prices[i])（j为偶数时为-，奇数时为+）。初始化DP[0][j] = -prices[0] / 0（j为偶数时为-prices[0]，奇数时为0）
- 可买卖多次，中间有冷却一天。思想：画状态转移图，有3个状态：持有/冷却/不持有。DP[i][0] = max(DP[i-1][0], DP[i-1][2]-prices[i]), DP[i][1] = DP[i-1][0]+prices[i], DP[i][2] = max(DP[i-1][2], DP[i-1][1])。对于冷却状态的初始化为0（即当天买卖后冷却，利润为0）。最后DP[len(prices)-1][1]与DP[len(prices)-1][2]的max即为返回值（即因负利润未买入（不持有）/ 最后处于冷却期）
- 卖出时有手续费。思想：同理，2个状态。只是在卖出时，额外-fee。初始化DP[0][1] = 0（虽然第零天买需要扣除手续费，但由于DP定义为最大利润，第1天持有时的利润将由第0天未持有时的利润推导得到，因此第零天不会卖出，即不应初始化为-fee），DP[0][0]=-prices[i]，手续费会在第一天及之后的状态转移过程中减去，无需在初始化时减去。

子序列问题 [300 674 718 1143 画二维表] 52 392 115 583 72 [647 由特征给出推导公式]

连续子序列，也称子数组

- **最长递增子序列。**思想：定义DP[i]为以nums[i]结尾的最长递增序列元素个数。计算DP[i]时，分别与之前的最长递增序列比较：如果nums[i]>nums[j]，则DP[i]=max(DP[j]+1, DP[i])，0<=j<i。初始化：DP[k]=1。最后返回max(dp)。

  方法1：DP O(N^2)；方法2：贪心+二分查找 O(NlogN)；方法3：二叉索引树：O(NlogN)

  应用 960：定义DP[i]为以s_k[i]为末尾的最长连续递增序列长度，即转移条件从原唯一字符串的s[j] <= s[i] 变为对于所有字符串的 all(s[j] <= s[i] for s in strs)

  优化O(nlong)：贪心+二分查找。定义D[i]为长度为i的递增序列中最小的末尾数。初始化D[i] = [nums[0]]，每次查找nums[i]在D[i]中的位置idx，表示了当前该递增序列的第idx位为nums[i]。若idx >= len(D)，则append其值，否则，更新D[idx]=nums[i]。

  应用 354：俄罗斯套娃[w, h]，将w从小到大排列，h从大到小排列，即转换为了对于h的最长递增子序列问题。若是相同w，则会覆盖，若是不同w，根据排序h>d[-1]时，将进行append（根据排序，说明当前到了下一个w）（即使下一个w的h小于之前w的h，说明之前w的h过大，也会将其覆盖）
- **最长递增子数组。**思想：定义DP[i]为0~i中最长连续递增序列元素个数。计算DP[i]时，只需与之前相邻的最长递增序列比较：如果nums[i]>nums[i-1]（即j只为i-1），则DP[i] = DP[i-1]+1。初始化：DP[k]=1。（此题也可贪心实现）

  方法1：DP O(N)；方法2：贪心+双指针 O(N)；方法3：分治 O(NlogN)；方法4：前缀和 O(N)
- **最长公共子序列。**思想：定义DP[i][j]为text1[0~i]与text2[0~j]中的最长公共子序列元素个数。当text1[i]==text2[j]时，DP[i][j] = DP[i-1][j-1]+1；否则，DP[i][j]=max(DP[i-1][j], DP[i][j-1])，DP[len(text1)][len(text2)]即为所求。（应用：两个字符串的删除操作 583，改题也可直接DP定义，即当text1[i]!=text2[j]时，DP[i][j]=min(DP[i][j-1]+1, DP[i-1][j]+1, DP[i-1][j-1]+2)）

方法1： DP O(N)

- 最长公共子数组。思想：根据直接的推导关系来定义DP，DP[i][j]为nums1[0~i]与以num2[0~j]中以nums1[i]与nums2[j]为结尾的连续子序列的长度。当nums1[i]==nums2[j]时，DP[i][j] = DP[i-1][j-1]+1。由于子数组是连续的性质，不同于上题，当nums1[i]!=nums2[j]时，DP[i][j]不与DP[i-1][j]或DP[i][j-1]存在关联（可画二维表观察规律）。初始化DP[0][0] = 0。max(DP)即为所求。（应用：不相交的线 1035）
- 最大子序列和。思想：dp[i] = max(dp[i-1]+nums[i], nums[i])
- 判断是否是子序列。思想：当成最长公共子序列做，若得出的长度为其中一个序列的长度，则是子序列。
- 子序列总个数。思想：dp[i][j]定义为num1[:i]与nums2[:j]中，nums1子序列为nums2的这样的子序列总数。分开讨论，即使nums1[i]==nums2[j]时，也假设不匹配将其计数（以纳入所有满足条件的子序列个数），dp[i][j] = dp[i-1][j-1]（匹配时）+dp[i-1][j]（不匹配时）；nums[1]!=nums2[j]时，dp[i][j] = dp[i-1][j]（不匹配）。最后dp[-1][-1]即为所求。
- 编辑距离。思想：画二维表观察增、删、改如何使两个string相同。dp[i][j]定义为num1[:i]与nums2[:j]之间最短的编辑距离。当nums1[i]==nums2[j]时，DP[i][j] = DP[i-1][j-1]；当nums1[i]!=nums2[j]时，有word1增加一个元素、word1删除一个元素（即word2增加一个元素）、word1与word2间替换一个元素：即DP[i][j] = min(DP[i][j-1], DP[i-1][j-1], DP[i-1][j])+1（直接话二维表可得）。初始化DP[0][k]=k, DP[k][0]=k。
- 回文子串个数。思想：本题应基于回文串的特征，给出推导公式：若nums[i, j]是回文串，且nums[i-1]==nums[j+1]，则nums[i-1,j+1]也是回文串。定义数量不直接，因此，DP[i][j]定义为nums[i, j]是否是回文串。当nums[i]!=nums[j]时，nums[i][j]不为回文子串，即DP[i][j]=False；nums[i]==nums[j]时，nums[i][j] = DP[i-1][j+1]，其中，若i=j或i+1=j，根据回文子串定义，DP[i][j]直接为True。该题不同于其他题的满二维表，其二维表是一个右上三角形，因此由下到上遍历行。
- 最长回文子序列长度。思想：本题应基于回文子序列的特征，给出推导公式：若num[i, j]是i至j最长回文子序列长度，且nums[i-1]=nums[j+1]，则nums[i-1, j+1]=nums[i,j]+2。因此，定义DP[i][j]为nums[i][j]中最长回文子序列长度，当nums[i]!=nums[j]时，DP[i][j]=max(DP[i+1][j], DP[i][j-1])；nums[i]==nums[j]时，DP[i][j] = DP[i-1][j+1]+2。该二维表也是右上三角形。注意i=j时，最长回文子序列长度为1；i+1=j且nums[i]==nums[j]时，最长回文子序列长度为2。初始化为1。
- **判断w是否为s的子序列：**

    1. 哈希表+二分查找 O(|w|* logN)，其中s长度为N，适用于有多个w的多次判断。 2. 双指针法：依次遍历，只能适用于单次判断 O(max(N, |w|))

记忆化存储

- 1444 思想：DP[i][j] = ∑(DP[x][j]+DP[i][y]). DP数组的更新在函数中进行，返回值为该DP[i][j]， 通过记忆化存储取代了DP数组
- 对于不同全局参数的记忆化存储，即需要用两次cached dfs，此时需要清空，使用dfs.cache_clear()

应用

- 741 1463 摘樱桃 思想：不能简单的视作从(0,0)到(n-1,n-1)以及从(n-1,n-1)到(0,0)的两次 先后 dp（因为每次最优的只是对于分段而言，而是应该整体而言，即前后同时出发。因此，状态空间定义为[k][i][j]，表示各走k步，从正向沿x方向走i，走到了[i][k-i]，逆向沿x方向走j（根据所求，也视为正向走），走到了[j][k+j]。dp[k][i][j]=max(dp[k-1][i-1][j-1], dp[k-1][i-1][j], dp[k-1][i][j-1], dp[k-1][i][j])。最后dp[-1][-1][-1]即为所求。

数位DP（统计数字问题）

通过变量传递，包含idx（必需），limit(必需，记录从0遍历到9或d[idx])，val(idx遍历到小于0时的返回值)，zero(连续前导0)，last(记录上一变量)，s(记录当前方案的list)等变量

- 233 思想：存储每个数位到数组d中（低位到高位的顺序）。ans=0; ans += dfs(idx, cnt, limit); return ans。通过参数cnt的传递，从搜索起点向下搜索到最底层得到方案数，一层层向上返回答案并累加，最后从搜索起点dfs(len(d)-1, 0, True)得到最终的答案。对于每一数位d[idx]，遍历的i是从1~9（必要时即limit=t时设置从1~该数位d[idx]），表示当该数位为i时，后续数字位的方案总和。i为目标数字时，参数cnt+1。并使用记忆化存储。相似题：600，788，902
- 其中涉及的位操作，如用s记录0~9的出现情况，可用 s|(2 << i) 实现第i位的记录，用(s >> i) & 1实现第j位的判断

游戏Game问题

思想：定义DP为先手者可获得的最大值，则dp可由本次元素与 (所有值之和-剩下元素对后手的最大值) 推导而得。

状态压缩 1915

对于有多个状态，每个状态上非0即1的问题，用状态压缩即二进制表示状态，通过异或更改相应位的状态，在DP/DFS中减小时间复杂度和空间复杂度。与前缀和搭配使用：

- 应用 1915 1371：关于求对于一个字符串，子串中各字符出现次数为odd的字符个数<=1的子串个数问题，利用状态压缩外，子串问题通过了前缀和的O(N)求解，即若state曾出现过，则从i到j的子串中出现次数为奇数的字符个数一定为0，对于出现次数为odd的个数为1的子串，其等于先对当前状态各位分别异或一次，分别求其曾出现的个数，即出现次数为odd的个数为even + 1 即odd。（类似前缀和思想，即若state[j] = state[i]，则[j+1: i]出现次数为奇的字符数是0）

图论

表示方式

邻接表：List[List[], ..., List[]]（size=N×K），更常用一些

邻接矩阵：List[List[], ..., List[]]（size=N×M）

思想

图是特殊的树，本质是多叉树的遍历（DFS），额外多了visited记录已遍历节点和已走路径防止成环（即防止重复遍历）。

- 类比于树的None为结束，即if not node: return ;
- 图以vis过为结束，即if vis[node]: return

图遍历框架：

```
def traverse(graph, s):
    if visited[s]:
        return
    visited[s] = True # 经过节点 s，标记为已遍历
    onPath[s] = True # 做选择：标记节点 s 在路径上
    for neighbor in graph.neighbors(s):
        traverse(graph, neighbor)
    onPath[s] = False # 撤销选择：节点 s 离开路径
```

- visited: 贪吃蛇已走过的节点。表示已遍历的节点。用于减小搜索空间，控制树遍历算法在图中的及时停止。
- onPath: 贪吃蛇的蛇身。表示当前走上的路径，和回溯实现比较相似，只是回溯是在for内执行，而onPath是在for外执行，以把第一个节点涵盖在内。用于判断路径是否有环，防止进入环中。
- `visit` 数组是以图为维度来考虑的（检测只遍历图中每个节点一次），`onPath` 数组是以路径为维度来考虑的（检测某一路径是否无环，进而得出图中是否无环）

当不存在环时，例如有向无环图，则不需要维护onPath变量。只需通过vis控制每个节点只访问一次即可。

问题分类：

基础遍历、拓扑排序（环检测）、并查集、最小生成树、最短路径、搜索问题（BFS）

797 遍历图（有向无环图）

思想：按模板即可，其中路径上的节点依次append进onPath。每当遍历到最后一个节点时，append当前的Path到结果中。

对于BFS算法，记得要一起处理当前que中的节点，然后一并加入que。因此visited的更新一定要放在for循环内部，记得也更新首个节点。

onPath是定义为[]还是定义为[False]*len(graph)，取决于需求，选当前需求下时间复杂度最小的。

图遍历例题：133, 399, 200, 127(转换为set后比list更高效，有时无需构图，只需拿到各自边集即可)

最小高度树 310 思想：BFS，每次砍掉度为1的节点（即叶节点），当队列len小于等于2时，队列中的结果即为返回值。

图四周：for a, b in pairwise((-1, 0, 1, 0, -1)): x, y = i + a, j + b

DFS遍历在有环图中的问题

对于DFS解决有环图中的问题：可分为两类：

1. 最值问题：不能解决，DP也无法解决，应该使用BFS解决
2. 路径存在问题：可解决

关于最值问题，由于DFS在遍历过程中需要标记visited与onPath，每个节点只允许遍历一次，而该节点可能走的并不是最值路径，其记忆的返回值并非最优值，由于无法二次更改，否则会造成访问节点而超两次，因此DFS无法解决有环图中的最值问题。

而关于路径存在问题，由于路径存在即可，而非像最值问题中的特殊的一条，因此DFS遍历每个节点的过程中，是任选一条可行路径，若无路径则说明其一定不存在到终点的可行路径。因此路径存在问题则可以通过DFS求解。

694 遍历顺序序列化

思想：根据方向赋予不同字符，需要根据去向（1~4）（即在调用前加上字符）以及撤销（-1~-4）（即在调用后加上字符）进而序列化

207 有向图环检测（是否存在环，即是否有一条路径上有环）

思想：(DFS版本) 往往对于环检测，是在路径维度上，只要该条路径上有环，则该图上则有环。因此，该题需要检测是否存在一条有环的路径，按照遍历模板，当onPath[node]=True时，说明当前路径存在环。

1559 无向图环检测

思想：DFS，依然通过onPath控制，但在dfs算法内取缔vis的限制（否则会矛盾），阻止访问回上一次的节点即可。

或施用并查集，连接各边时若其以属于同一并查集，则该图存在环。

211 拓扑排序

思想：(DFS版本) 有环图无法进行拓扑排序。拓扑图是基于度数，High level上从左到右，入指向出。对后序遍历结果的反转，即拓扑排序的结果。首先判断是否有环，有环则返回[]；否则，返回后续遍历的reverse，后续遍历即是在for循环后，记录node。

为什么图DFS算法后序遍历的结果是拓扑排序：由后续遍历可知，一个节点只有在它所依赖的节点访问完才被访问，根据拓扑排序的定义，节点只有在依赖于它的节点被访问完才被访问。因此，后续遍历的reverse即是拓扑排序的结果。

(BFS) 版本对入度为0的节点依次加入队列，每次从队列中pop，其指向的节点入度-1，更新队列。若无节点可pop，但未遍历一遍，则说明有环。无环时，节点的pop顺序即为拓扑排序结果。相比于DFS，BFS的拓扑排序可以输出所有到某一顺序的序列。其实现即是在BFS的基础上，相比于DFS，额外增加记录度数，通过tmp数组将顺序相同的一并添加到BFS的que中。

二分图

二分图模型举例：「文章」与「题目」是多对多关联。只要查询任意一个「文章」节点即可得到关联的「题目」节点，查询任意一个「题目」节点，即可得到关联的「文章」节点。二分图同色之间不存在边。

785 判定二分图（双色问题）

思想：基于DFS算法的图遍历模板。只用visited记录图（即全局level）是否访问过。判断对象是邻接节点（即for循环内进行判断），若未访问过该邻接节点，则为该邻接节点赋色（与其当前节点不同的颜色）；反之，比较其颜色，若存在相同，则该图不是二分图。由于存在0度节点，因此需要对每个节点开展traverseG，之前过程中已visited的节点可跳过。

并查集

根据连通性进行判断。若两个节点连通（直接/间接），则其属于同一个集。通过树的结构实现，数组存储其父节点索引。过程中，find函数含有路径压缩，即u == father[u] ? u: father[u] = find(father[u])，使得find为O(1)；connected直接使用self.find(p) == self.find(q)，O(1)；union即若它们不相连，则将一个节点的根节点作为另一个节点的根节点的父节点，为O(1)；只有构造时，即self.parent = [i for i in range(n)] 为O(n)，每个节点的parent初始化为自己，即自环。

不同的集即一个连同分量。

```
def __init__(self, n: int): O(N)
    self.parent = [i for i in range(n)]
```

```
def union(self, p: int, q: int): O(1)
    rootP = self.find(p)
    rootQ = self.find(q)
    if rootP == rootQ:
        return
    self.parent[rootQ] = rootP // 连通分量个数-1

def connected(self, p: int, q: int) -> bool: O(1)
    rootP = self.find(p)
    rootQ = self.find(q)
    return rootP == rootQ

def find(self, x: int) -> int: O(1)
    if self.parent[x] != x:
        self.parent[x] = self.find(self.parent[x])
    return self.parent[x]
```

130 Surrounded Region(并查集应用)

思想：可以用DFS实现；或用并查集实现。创建独属于O的并查集，首先位于四周的O加入属于O的并查集，而再依次从第一行至第n行遍历，若其当前状态是O且其四周也是O，则将其加入其四周的并查集中（能保证即使只是临时的O，通过并查集也能与永久的O区分开，因为根据该算法只有永久的O在该并查集）。最后只将位于O并查集中的元素（遍历+connected）赋值为O, 其他为X.

990 Satisfiability of Equality Equations(并查集应用)

思想：将该题具有==关系的变量字母，union到一个并查集；然后依次遍历具有!=的式子，判断其是否connected，若connected，则说明出现冲突，返回False。

959 Regions Cut By Slashes(并查集应用)

思想：关键在于如何将斜线转换为图。通过将一个格子视为4个三角形进行实现，视斜线为划分连通的依据。进而根据连通分量个数确定并查集个数。

803 928 并查集中统计各连通分量中元素的个数

思想：在UF init中额外通过self.size记录各连通分量中元素的个数，当P union到Q时，即self.parent[rootP] = rootQ，则self.size[rootQ] += self.size[rootQ]。而通过self.size[self.find(x)]，即可找到x所属的连通分量中元素的个数

1697 并查集求路径上边的上界

思想：利用并查集求路径上所有边的上界。利用边权重作为并查集划分的依据。遍历路径，对排序好的edgeList中的边，依次check其权重并union。

生成树

包含所有图节点的无环连通子图（即图中所有点和若干边的集合）。最小生成树：该子图的边权重和最小。

261 以图判树

思想：添加的这条边，如果该边的两个节点本来就在同一连通分量里，那么添加这条边会产生环；反之不会。因此加入边前进行Union.connected判断能够确保图不包含环（即是树）。

生成最小生成树的逻辑：1.是一棵树 2.包含所有节点 3.权值尽可能小

1584 求最小生成树（Kruskal 克鲁斯卡尔算法，即加边法）

思想：基于并查集与贪心思想（直接全部排序或堆）。且确保不会形成环和森林（即非连通图）。对边按照权值从小到大排列（或每次通过堆取一个最小权值的边），若当前边的左右端点是connected的，说明不能加该条边（否则会成环）。反之，将该边加入至最小生成树，同时计入权值。（当uf.count !=1时，说明图是非连通图）

时间复杂度：主要是边的权值排序，需O(ElogE)，并查集的构造为O(V), 判断为O(1)

求最小生成树关键边：去掉该边，图不连通或最小生成树权值变大；最小生成树非关键边：提前连通该边所连通的节点，最小生成树权值=新最小生成树权值+该边权值。

求最小生成树（Prim 普莱姆算法，即加点法）

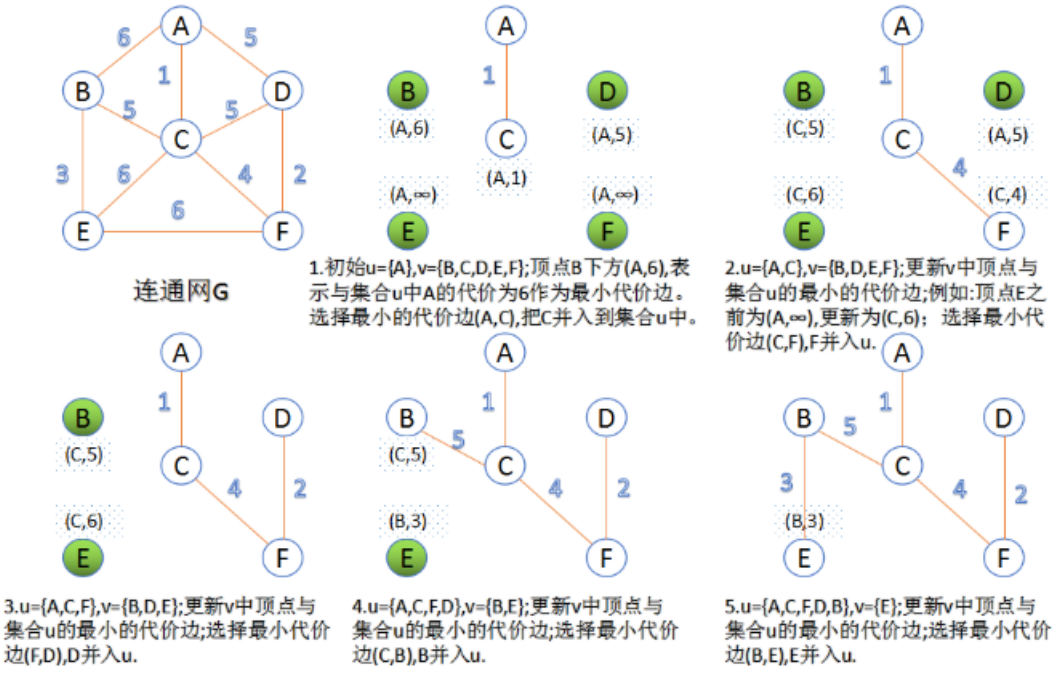
思想：基于BFS、优先级队列（也可用heapq实现）（import queue, pq = queue.PriorityQueue(), pq.put((key, val))）、贪心思想。通过切分（一刀把图中节点分为两个非空集合）实现，每一次切分一定可以找到最小生成树中的一条边，权重最小的横切边（被切中的边）即构成最小生成树的一条边。O(ElogE)

每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个顶点s开始，逐渐长大覆盖整个连通网的所有顶点：

1、图的所有顶点集合为V；初始令集合u={s},v=V−u;

2、在两个集合u,v能够组成的边中，选择一条代价最小的边(u0,v0)，加入到最小生成树中，并把v0并入到集合u中。

3、重复上述步骤，直到最小生成树有n-1条边或者n个顶点为止。由于不断向集合u中加点，所以最小代价边必须同步更新



743 求最短路径 Dijkstra算法

思想：基于BFS与小顶堆。以节点为单位，将路径长度作为小顶堆的排序依据，即首先把开始节点入堆，距离数组dists中，其距离设为0（其余为-inf），依次遍历其相邻节点。若算出到该节点距离+边权重 > 已知到该邻接节点距离 （此步根据题意，如1631 最短路径 可能是max(节点距离,边权重) > 已知到该邻接节点距离；1514 大顶堆），则更新节点距离，将邻接节点入堆，直至堆为空。通过heapq实现（import heapq, pq = [], heappush(pq, (k, v))插入，heappop(pq)弹出，默认根据k从小到大排序，从大到小在k前加负号即可），key中存储当前路径上的值 [当前节点序号, 当前值]，dists数组存储最优值。

剪枝：

- 节点路径长度小于其最优路径长度，即dis<dists[node]，说明一定不会选择该条路径，直接continue
- 若只算到某个节点的距离，当pop出该节点时，即可return（因若pop出了该节点，说明已无可更新该节点的节点）

时间复杂度O(ElogV)

若路径无权重，可直接使用BFS求解。

带限制的最短路径（1293）：可突破k次网格。思想：与BFS基本算法求解相同。额外对visited扩展维度，来记录当前突破了几次限制。因根据最短路径即最优的思想，限制突破次数越少越好，每次对visited[i][j][:x]更新为True（因visited[i][j][k]=True时后续无法访问，即代表此前已有更优情况）

搜索问题 847

BFS搜索

思想：搜索问题首先分析状态。即搜索问题的关键是对于状态空间的定义，常定义于visited/存入que中，大多基于BFS算法求解。若不允许重复访问节点，则根据BFS性质，step可以不存放于状态中，在for循环外增加即可。根据BFS性质，第一个更新visited数组状态对应的路径是最短的。

- 若想从多个节点开展搜索，在初始时将所有节点加入que中即可，注意其各自state的区分
- BFS搜索尽量拆分成子序列搜索，因为搜索序列越长，栈堆积越大，越容易超时。
- BFS可实现优先队列搜索，即每次对值小的执行que.appendleft，注意此时要取消for循环。之前执行for _ in len(que)循环是为了保证广度优先方便统计depth（之前没有for循环也可以，但对于优先队列必须取消for循环，否则优先队列将没有意义），此时visited只对pop的对象，即邻域对象前进行检查与标记（因对于普通队列的BFS保证在邻域for循环内更新visited可行，但对于更general的情况，如优先队列，则必须在邻域for循环外更新）
- visited数组多定义为set()，三维以上数组情况较少出现。

双向BFS：主要用于提高搜索效率。从起点和终点同时出发，若中间结点重合，则找到了路径。

对于找最小路径，BFS搜索比DFS搜索具有更高的效率。

DFS搜索

1.基本题思想：常只记录vis数组（可通过改变grid来省略时间复杂度与连带的时间复杂度），通常直接对本节点进行条件判断（如大小范围、vis等），而非for循环中的邻接节点（这样可保证可对所有节点开展遍历）。可对所有坐标进行搜索。

2.将n个数分为k个和相同的数组：思想：基于DFS，对每个数进行处理，即每个数必须分在这k个数组之中。结束条件：匹配到最后一位时，即index == len(nums) 说明匹配成功，则返回True (因为根据算法，此时所有数一定在k个数组里面，即一定满足了条件)。同时需要将数组从大到小排列（大的在前，不能hold的情况将更快出现，可以更快结束递归）。DFS基本一致，主要是通过剪枝来降低时间复杂度。剪枝方式：

- state[i] == state[i-1]，由于上一状态匹配失败才会落入下一状态i，因此可直接continue。
- 当都没有匹配成功时，即cur[i]=0，则直接end（因为若当前的集合不能hold这个数, 其他集合也同样不能hold它) (当max(nums) > div，说明存在不能hold的数)

更General情况：当分到的和允许不相同时，改变停止条件即可，停止值为当前已找到的最小值。

3.记忆化搜索，保存函数结果，即第(i,j)处的最优值，其周围的值可根据该最优值推导。（要确保最优值不会变化）和动态规划有一定关系。

- 例题 2328

  思想：对DP进行定义，DP[i][j]即为以(i,j)起始的递增数列个数。DP[i][j] = ∑(DP[adj_i][adj_j] + 1)。由于递增数列不可逆，因此实现过程中不会重复递归，保证了该转移公式在不给定特殊条件即可处理

定义DFS的自变量应该从状态入手，无论是否可重复访问，若设置了合理的状态，则每个状态都是不同的。如1575，由于状态包括[位置、剩余油量]，因此定义dfs(index, left)

图拷贝

思想：设计traverse(node)函数，返回值为拷贝后的node，对其子节点依次调用，并将其返回值作为拷贝节点的子节点。由于图中节点可能会被访问两次，因此利用dic存储节点对应的拷贝节点，若其已访问过，则直接返回dic中的拷贝节点。

数学问题

随机算法

1. 洗牌算法 384

   思想：打乱数组中数字顺序。遍历索引，利用randint产生随机数idx: (i, len(nums)-1)，swap将idx位置上交换过来到i。进而使得已遍历位置都以相同概率被确定（n-1/n * n-1/n-2 * ... * 1/i = 1/n)，未遍历处都是未被选中的数字。

2. 抽样算法 382

   思想：从链表或数组中随机抽取一个数。对于索引i上的数，有1/i概率（通过randint(1, i) == 1实现）选中作为返回值，未选中，则保留上个选中的数作为返回值。进而使得所有数字都有1/n概率被选中。（1/i概率选中，i/i+1 * i+1/i+2 * ... * n-1/n概率被后续保留，因此总概率为其乘积即1/n）

   $$\begin{aligned}&\frac{1}{i}\times(1-\frac{1}{i+1})\times(1-\frac{1}{i+2})\times\ldots\times(1-\frac{1}{n})\\=&\frac{1}{i}\times\frac{i}{i+1}\times\frac{i+1}{i+2}\times\ldots\times\frac{n-1}{n}\\=&\frac{1}{n}\end{aligned}$$

   延伸到抽k个数，则对于索引i上的数，有k/i概率选中作为返回值，取代已选定的k个数（初始化一下）中其中一个

   $$\begin{aligned}&\frac{k}{i}\times(1-\frac{k}{i+1}\times\frac{1}{k})\times(1-\frac{k}{i+2}\times\frac{1}{k})\times\ldots\times(1-\frac{k}{n}\times\frac{1}{k})\\=&\frac{k}{i}\times(1-\frac{1}{i+1})\times(1-\frac{1}{i+2})\times\ldots\times(1-\frac{1}{n})\\=&\frac{k}{i}\times\frac{i}{i+1}\times\frac{i+1}{i+2}\times\ldots\times\frac{n-1}{n}\\=&\frac{k}{n}\end{aligned}$$

位运算

1. 异或运算 (a^a=0) 136 268

   思想：用于求出现次数为1的元素。因为元素自身异或为0，任何元素与0异或为自身。因此将一个数组里的数进行连续异或，则可求出其中唯一的次数为1/或缺失的元素。异或运算满足交换律。

   拓展：寻找[1,n]数组内重复元素：空间复杂度 O(1)。思想：在原数组内操作，通过修改索引为其值处的值为负来作标记，若标记前其索引为其值的值已经为负，则说明此元素重复。寻找[1, n]数组内消失元素：空间复杂度 O(1)。思想同上，最后数组中值大于0的值对应的索引+1即为消失元素的值。

2. 与运算 (n & (n-1)) 191

   思想：用于求二进制中1的个数。n & n-1可把二进制中最后一个1变为0。因此，通过该不断使n = n&n-1至n为0，即可求解1的个数。

3. 与运算 (x & -x)

   思想：常用于二叉索引树，用于获取二进制最低位的1所对应的十进制的值。

4. 掩码 672

   思想：设置二进制掩码，表示只想在特定位上实现操作。二进制掩码可通过mask=0b110110来设置。对于n位，则首先创建二进制字符串，而后用int(x,2)将二进制字符串x转换为二进制数，如mask = int('10' * (n // 2), 2)。将x以二进制输出，使用print(bin(x))

   二进制左移n位并保证位数m不变：(mask << n) & ((1 << m) - 1)

5. 二进制数加法 67

   思想：二进制数a + b即a ^ b ^ carry。implementation时，视a ^ b为新的a，a & b的carry为新的b，连续异或，直至b即carry为0：while b: a, b = a ^ b, (a & b) << 1（a，b先转为int(a, 2)与int(b, 2))

对于二进制的前31位，将二进制转为十进制，若该位为1，ret += 1 << i；第32位是符号位，处理该位时，若该位为1，ret -= 1 << i.

阶乘算法

思想：n! 中后导0的个数，即从1到n中含有因子5的数量。因此对于i从1到log5(n)，n//(5**i)之和即为后导0总个数。

素数算法

思想：寻找小于n的素数。则设置一个标记是否为素数的数组（初始化为True），若i为素数，则将i的倍数标记为非素数(for j in range(i*i, n, i))。时间复杂度O(根号(N)·logN·logN)

```
for i in range(int(math.sqrt(n))+1):
    if isPrime[i]:
        for j in range(i*i, n, i):
            isPrime[j] = False
```

模除运算

1. (a * b) % k = (a % k) * (b % k) % k：对乘法求模，等价于先对每个因子求模，然后对相乘结果再求模。
2. (a + b) % k = ((a % k) + (b % k)) % k：对加法求模，等价于先对每个加数求模，然后对相加结果再求模

快速求幂算法

思想：求x^k，将O(k)的求幂优化到O(logk)，k为指数：

$$a^b = \begin{cases} a \times a^{b-1}, b \text{ 为奇数} \\ (a^{b/2})^2, b \text{ 为偶数} \end{cases}$$