八股文笔记

> 作者： Wang Luning，转载请注明出处

LeetCode

> 主要基于hot100。

基本数据结构

- `list`

  可以当成栈使用：`ls = []`

  - 入栈：`ls.append(<ele>)`
  - 出栈：`ls.pop()`，返回出栈的元素

  也可以当成队列用：

  - 将`x`插入到队列的第i个位置：`ls.insert(i, x)`
- `str`
  - 找到第一个出现字符x的位置，若没有该字符则返回-1：`str_.find(x)`
  - 计算字符串中字符x出现的次数：`str_.count(x)`
- `dict`

  可以当成哈希表使用：`dic = dict()`

  - 删除元素：`dic.pop(<key>)`
  - 加入元素/修改元素：`dic[<key>] = <value>`
- `set`

  可以用于判断重复元素：`set_ = set()`

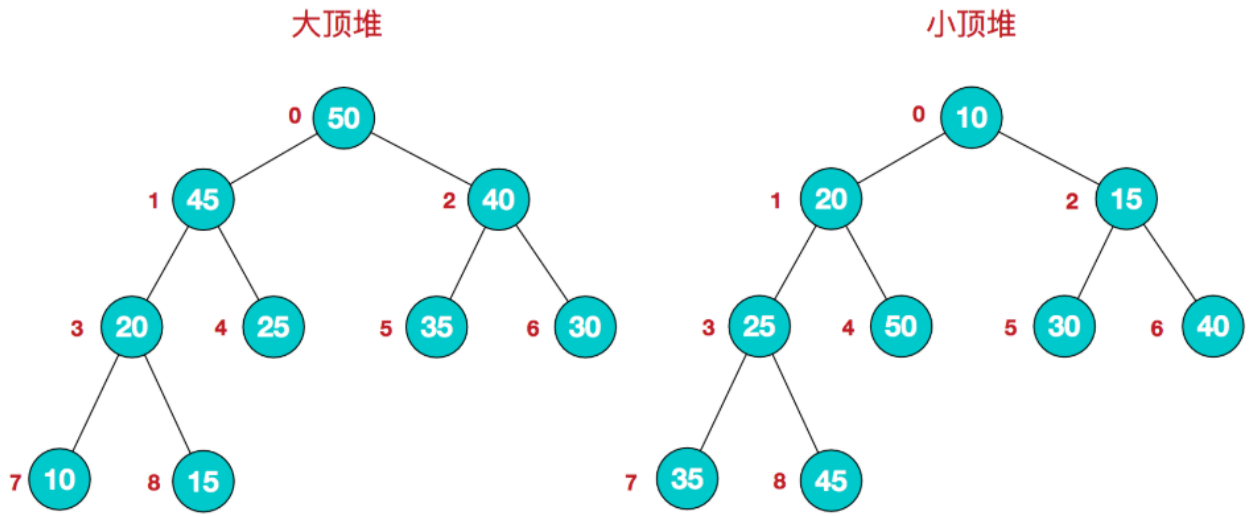
  - 加入元素：`set_.add(<ele>)`
  - 删除元素：`set_.remove(<ele>)`
  - 判断某个元素是否在集合中：`<ele> in set_`
- `collections.deque`

  双端队列：`queue = collections.deque()`

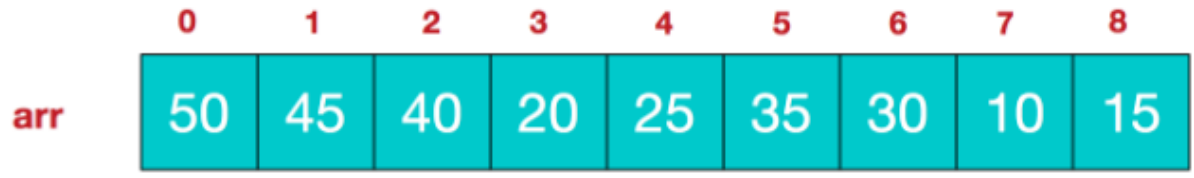
- 左端入队：`queue.appendleft(<ele>)`
- 左端出队：`queue.popleft()`，返回出队的元素
- 右端入队：`queue.append(<ele>)`
- 右端出队：`queue.pop()`，返回出队的元素
- 查看左端元素：`queue[0]`
- 获取队列长度：`len(queue)`

- `heapq`

  堆的本质是一种**完全二叉树**，其确保每个节点都大于等于其左右孩子（大顶堆）或小于等于其左右孩子（小顶堆）



  对于堆中的节点按层进行编号（设根节点的index为`0`），映射到数组中即为：



  该数组从逻辑上就是一个堆结构，其定义为：

  - 大顶堆：`arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]`
  - 小顶堆：`arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]`

  也即，每个节点的值都大于等于/小于等于其左右子节点的值

  这个数组本质上就是优先队列（**默认为小顶堆**），其堆顶元素总是最小值。其操作对象为普通列表，是对列表的原地操作（作用完之后其仍是普通列表，只不过是能确保其元素排列满足堆的要求）。python中使用`heapq`来进行优先队列操作：

  - 将元素添加到堆中：`heapq.heappush(heap_list, n)`
  - 弹出堆顶，并将堆顶换为新的最小值：`heapq.heqppop(heap_list)`
  - 将一个可迭代对象（如列表）原地转换为堆：`heap.heapify(heap_list)`

  例如：

```python
import heapq

heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 4)
heapq.heappush(heap, 2)
print(heap)  # 输出: [1, 2, 4, 3]（堆按照最小堆特性排列，确保堆顶元素1是最小元素）

heapq.heappop(heap)  # 弹出最小值 1
print(heap)  # 输出: [2, 3, 4]

nums = [4, 3, 1, 2]
heapq.heapify(nums)
print(nums)  # 输出: [1, 2, 4, 3]（转换为最小堆）
```

如果要构造大顶堆的话，可以通过将列表元素转为负数然后使用`heapq.heapify()`构造小顶堆实现，此时堆中存的元素都是原始列表元素的相反数，因此`heappush`和`heappop`方法也要相应地改变，push和pop时都要带上一个负号。将它们封装为一个自定义类后，从外部看来就是一个大顶堆：

```python
import heapq

class heapq_max:
    def heapify(self, nums):
        """将列表转换为大顶堆"""
        max_heap = [-x for x in nums]
        heapq.heapify(max_heap)
        return max_heap

    def heappush(self, heap, item):
        """向大顶堆添加元素"""
        heapq.heappush(heap, -item)

    def heappop(self, heap):
        """从大顶堆弹出最大元素"""
        return -heapq.heappop(heap)
```

当输入heap的不是数值时（例如每个元素是一个元组），那么推入堆时依据的是首个元素的大小，如果首个元素不同再依次比较后边的元素。例如：

```python
import heapq

heap = []
heapq.heappush(heap, (3, 'a'))
heapq.heappush(heap, (1, 5))
heapq.heappush(heap, (4, 5, 7))
heapq.heappush(heap, (2, 'mn'))
print(heap)  # 输出: [(1, 5), (2, 'mn'), (4, 5, 7), (3, 'a')]
```

对于更复杂的自定义数据结构（如链表节点），如果想将其放在堆中，可以通过给数据结构本身添加`__lt__`方法，来定义对象之间比较大小的逻辑，这样就可以利用堆来进行排序了。例如：

```
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next

ListNode.__lt__ = lambda node1, node2: node1.val < node2.val

heap = [ListNode(1), ListNode(2)]
heapq.heapify(heap)
```

将无序列表转化为（大顶）堆的手动实现：

- 首先，由于列表构成的一定是完全二叉树，因此列表某个位置及之前的节点一定都是非叶子节点，后边的位置全都是叶子节点。通过一些推导可知：**最后一个非叶子节点在列表中的索引为`len(nums)//2 - 1`**。同时，也知道索引为`i`的节点的左右子节点（若存在）在列表中的索引分别为`2*i+1`与`2*i+2`（设根节点索引为`0`）
- 算法流程：

  从最后一个非叶子节点（`nums[len(nums)//2-1]`）开始，从后往前遍历各个非叶子节点，每次都将以这个节点为根节点的子树转为一个大顶堆（也即从下往上依次收拾清各个非叶子节点的子树）。设`max_heapify(nums, n, i)`函数将数组中索引为`i`的（非叶子）节点的子树排成大顶堆，`n`为数组总大小（用于判断左右子节点是否存在于树的范围中），从后往前依次遍历每个非叶子节点，并将其子树转为一个大顶堆。

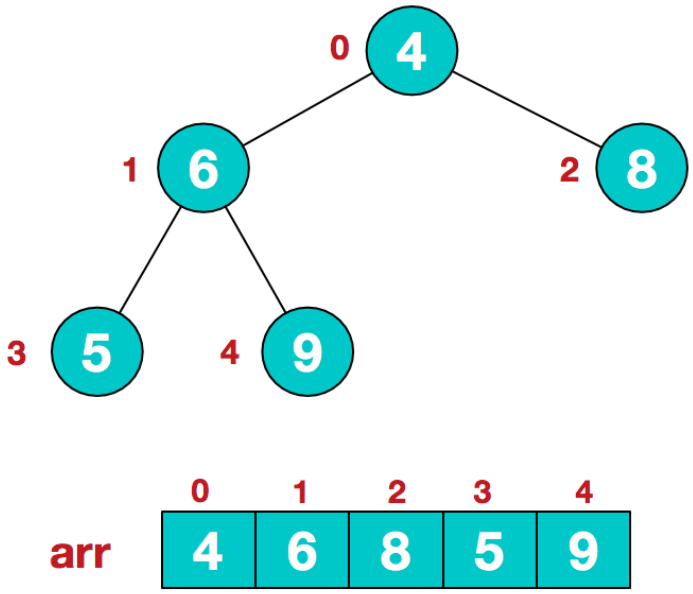
  具体而言，`max_heapify(nums, n, i)`函数中，首先检查一下当前节点的左右子节点（存在性通过`2*i+1<n`和`2*i+2<n`是否成立来判断）中最大的那个是否大于了当前节点，若发现大于的话（假设最大值的索引为`largest`）则将当前节点和这个最大的子节点的值进行互换，这样的话就可以保证交换后的`nums[i]>=nums[2*i+1], nums[i]>=nums[2*i+2]`，也即这里局部满足了大顶堆条件。进一步，还需确保交换后的`largest`节点（左右节点之一，具有原先`nums[i]`的值）的子树也是大顶堆（因为交换后可能会扰乱原先排好的`nums[largest]`子树的大顶堆结构），因此需要进一步递归地调用`max_heapify(nums, n, largest)`

  ```
  def build_max_heap(nums):
      n = len(nums)
      # 从最后一个非叶子节点开始向前调整
      for i in range(n // 2 - 1, -1, -1):
          max_heapify(nums, n, i)

  def max_heapify(nums, n, i):
      """
      调整以i为根的子树，使其满足大顶堆性质
      n: 堆的大小
      i: 当前节点的索引
      """
      largest = i  # 假设当前节点是最大值
      left = 2 * i + 1  # 左子节点
  ```

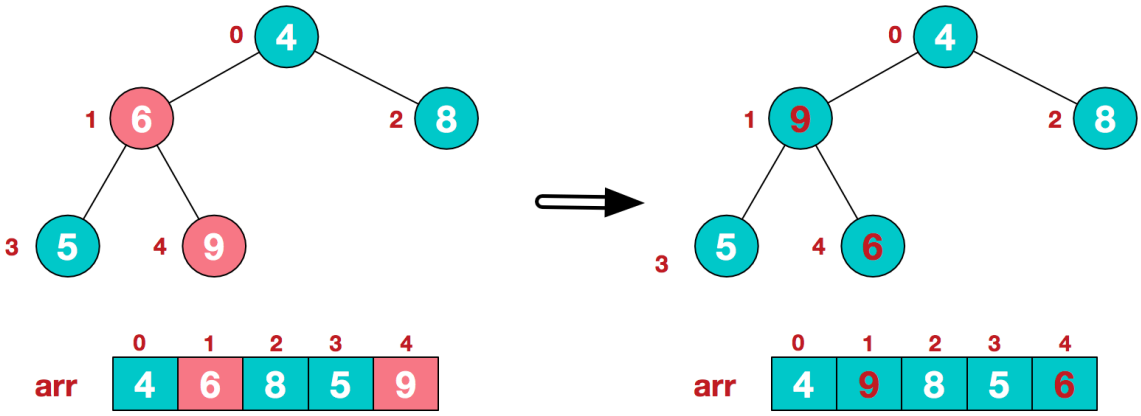
```
    right = 2 * i + 2  # 右子节点

    # 检查左子节点是否大于当前节点
    if left < n and nums[left] > nums[largest]:
        largest = left

    # 检查右子节点是否大于当前节点
    if right < n and nums[right] > nums[largest]:
        largest = right

    # 如果最大值不是当前节点，交换并继续调整
    if largest != i:
        nums[i], nums[largest] = nums[largest], nums[i]
        max_heapify(nums, n, largest)
```

示例：

假定给定无序数组如下：

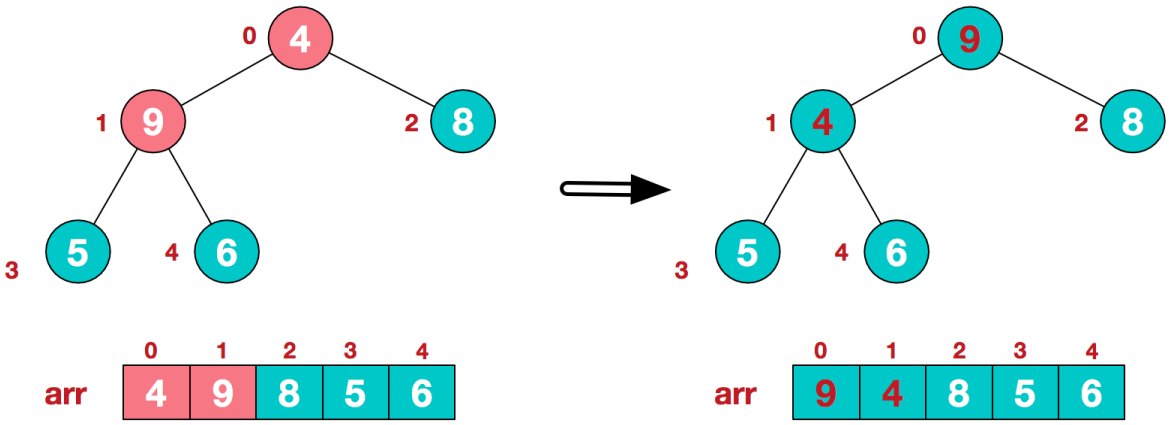


可见最后一个非叶子节点的索引为：`5//2-1=1`，因此也即`arr[1]=6`对应的节点。针对这个节点，发现其右子节点`arr[2*1+2]=arr[4]=9`大于6，所以将6和9这两个节点进行互换：

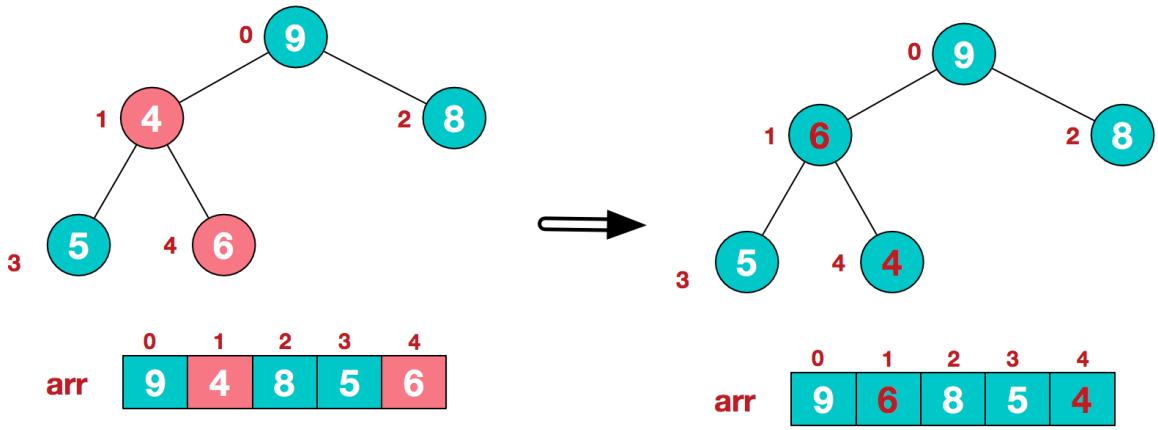


交换后还需进一步递归地将`nums[4]`的子树也变成大顶堆，但本例中`nums[4]`没有子树了，所以不用再继续递归了。

然后，将倒数第二个非叶子节点的子树变为大顶堆，也即`arr[0]=4`的子树。对比发现左节点`arr[1]=9`最大且大于`arr[0]=4`，因此将二者进行交换：



交换后还需进一步递归地将`arr[1]`的子树也变成大顶堆，可以看到交换后确实破坏了`arr[1]`子树原先的大顶堆结构，因此进一步对其调用`max_heapify`函数，发现`arr[4]=6`最大且大于`arr[1]=4`，因此将它们两者进行交换：



同样，交换后的`arr[4]`没有子树，因此不用继续递归造堆了。

至此，所有非叶子节点都处理完毕，此时的列表已经变成了一个大顶堆。

排序算法

	时间复杂度			空间复杂度	稳定性	就地性	自适应性	比较类
	最佳	平均	最差	最差				
冒泡排序	O(N)	$O(N^2)$	$O(N^2)$	O(1)	稳定	原地	自适应	比较
插入排序	O(N)	$O(N^2)$	$O(N^2)$	O(1)	稳定	原地	自适应	比较
选择排序	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(1)	非稳定	原地	非自适应	比较
快速排序	O(N log N)	O(N log N)	$O(N^2)$	O(log N)	非稳定	原地	自适应	比较
归并排序	O(N log N)	O(N log N)	O(N log N)	O(N)	稳定	非原地	非自适应	比较
堆排序	O(N log N)	O(N log N)	O(N log N)	O(1)	非稳定	原地	非自适应	比较
基数排序	O(Nk)	O(Nk)	O(Nk)	O(N+k)	稳定	非原地	非自适应	非比较
桶排序	O(N+k)	O(N+k)	$O(N^2)$	O(N)	稳定	非原地	自适应	非比较

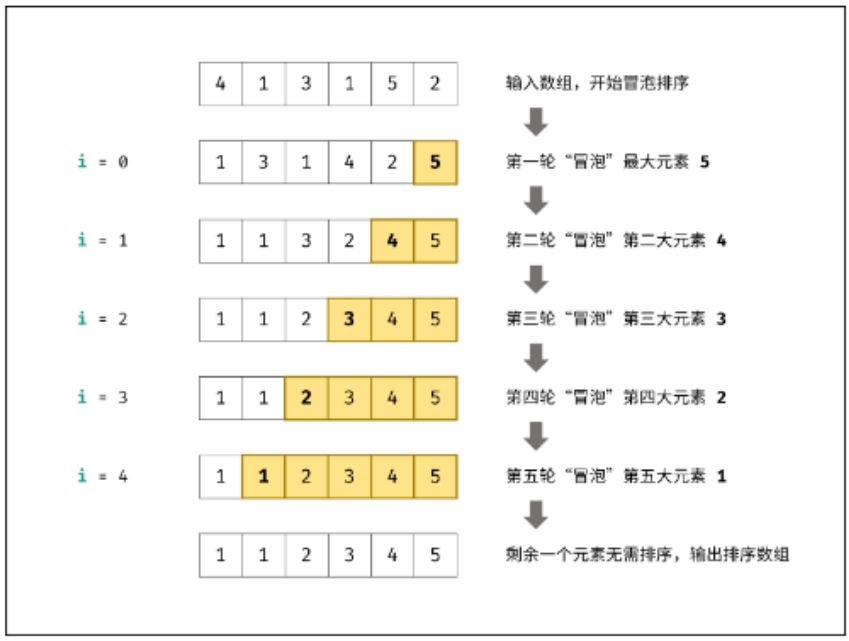
比较型排序

比较型排序依赖各个元素之间的比较，可以证明任何这种排序方法最坏情况下的时间复杂度都不低于$O(N \log N)$。

- **冒泡排序：**
  - 内循环： 使用相邻双指针 j , j + 1 从左至右遍历，依次比较相邻元素大小，若左元素大于右元素则将它们交换；遍历完成时，最大元素会被交换至数组最右边 。内循环共`N-i-1`轮。
  - 外循环： 不断重复内循环，每轮将当前最大元素交换至 剩余未排序数组最右边 ，直至所有元素都被交换至正确位置时结束。外循环共`N-1`轮。

```
def bubble_sort(nums):
  N = len(nums)
  for i in range(N - 1):              # 外循环
    for j in range(N - i - 1):    # 内循环
      if nums[j] > nums[j + 1]:
                # 交换 nums[j], nums[j + 1]
        nums[j], nums[j + 1] = nums[j + 1], nums[j]
```

（每轮剩下的未排序数组都处于左侧，每轮会让右侧已排好的序列长度+1）



普通冒泡排序最佳时间复杂度为$O(N^2)$，可通过标志位实现提前返回（当某一轮外循环中没有任何元素被交换，说明此时左侧剩下的未排序数组其实也已经排好），在原序列本身就是排好的情况下可优化到$O(N)$：

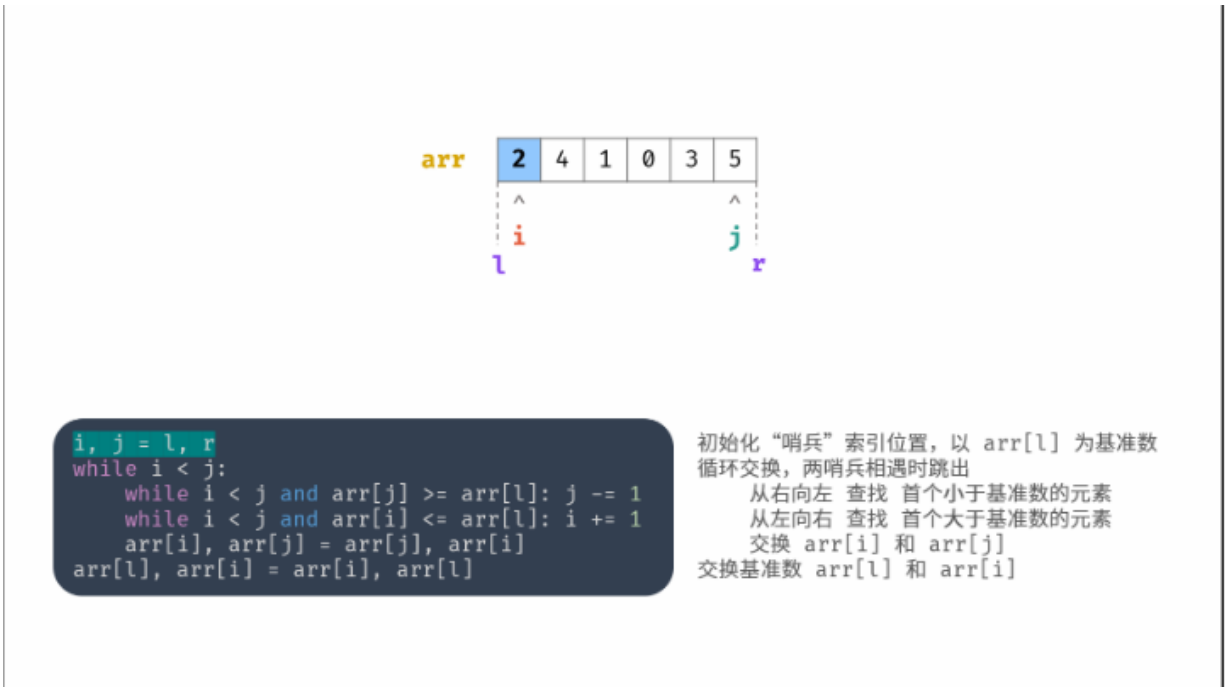
```
def bubble_sort(nums):
  N = len(nums)
  for i in range(N - 1):
    flag = False          #  初始化标志位
    for j in range(N - i - 1):
      if nums[j] > nums[j + 1]:
        nums[j], nums[j + 1] = nums[j + 1], nums[j]
        flag = True  # 记录交换元素
    if not flag: break   # 内循环未交换任何元素，则跳出
```

- **快速排序：**

  快速排序算法有两个核心点，分别为 **哨兵划分** 和 **递归**：

  - 哨兵划分：

    以数组某个元素（一般选取首元素）为 **基准数**，将所有小于基准数的元素移动至其左边，大于基准数的元素移动至其右边

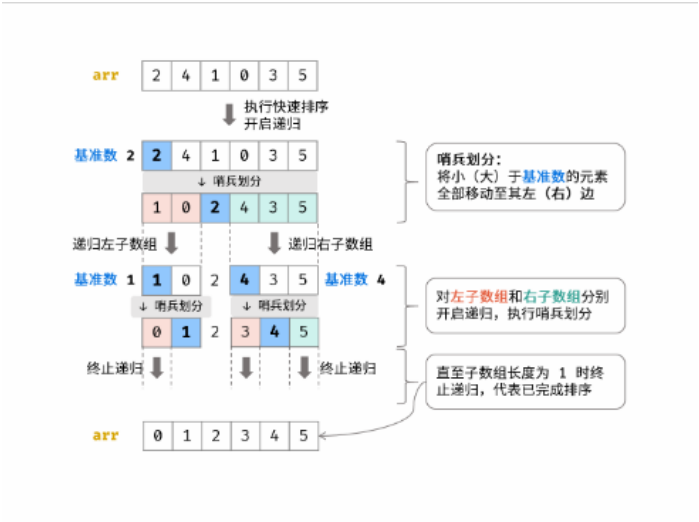



```
i, j = l, r
while i < j:
    while i < j and arr[j] >= arr[l]: j -= 1
    while i < j and arr[i] <= arr[l]: i += 1
    arr[i], arr[j] = arr[j], arr[i]
arr[l], arr[i] = arr[i], arr[l]
```

初始化“哨兵”索引位置，以 arr[l] 为基准数
循环交换，两哨兵相遇时跳出
    从右向左 查找 首个小于基准数的元素
    从左向右 查找 首个大于基准数的元素
    交换 arr[i] 和 arr[j]
交换基准数 arr[l] 和 arr[i]

- 递归：

  对 **左子数组** 和 **右子数组** 分别递归执行 **哨兵划分**，直至子数组长度为 1 时终止递归，即可完成对整个数组的排序。




```
def quick_sort(nums, l, r):
    # 子数组长度为 1 时终止递归
    if l >= r: return
    # 哨兵划分操作
    i = partition(nums, l, r)
    # 递归左（右）子数组执行哨兵划分
    quick_sort(nums, l, i - 1)
    quick_sort(nums, i + 1, r)

# 哨兵划分函数
def partition(nums, l, r):
    # 以 nums[l] 作为基准数
    i, j = l, r
    while i < j:
        while i < j and nums[j] >= nums[l]: j -= 1
```

```
            while i < j and nums[i] <= nums[l]: i += 1
            nums[i], nums[j] = nums[j], nums[i]
        nums[l], nums[i] = nums[i], nums[l]
        return i

    # 调用
    nums = [3, 4, 1, 5, 2]
    quick_sort(nums, 0, len(nums) - 1)
```

> **注意**：按照上述写法，每轮循环中必须先动`j`（`while i < j and nums[j] >= nums[l]: j -= 1`），再动`i`（`while i < j and nums[i] <= nums[l]: i += 1`），最后返回前交换`nums[l], nums[i] = nums[i], nums[l]`，才能确保结果正确。不能先动`i`再动`j`，否则最后会出现边界错位问题。

最佳情况下，每轮哨兵划分将数组划分为等长度的两个子数组，哨兵划分操作时间复杂度$O(N)$，递归轮数共$O(\log N)$，最佳总复杂度$O(N \log N)$

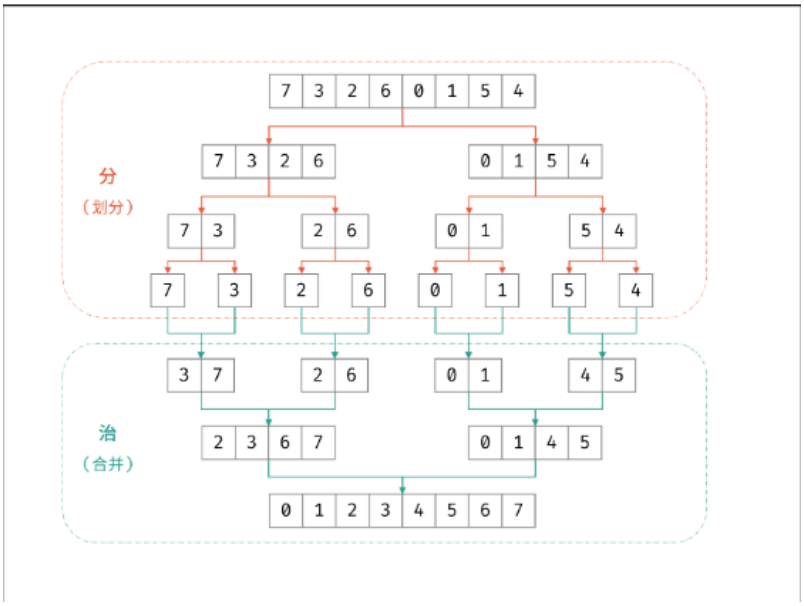
平均总复杂度：$O(N \log N)$

最差总复杂度：每轮哨兵划分操作都将长度为N的子数组划分为1和N-1的两段，此时递归轮数达到N，总复杂度$O(N^2)$

- **归并排序：**

  体现了分而治之的思想。先递归地将数组从**中点位置**分开，将原数组的排序问题转化为了子数组的排序问题；当划分到子数组长度为1时，开始向上合并，不断将**左右两个较短排序数组** 合并为 **一个较长排序数组**，直至合并至原数组时完成排序。时间复杂度为$O(N \log N)$，且它是稳定的，不随排序的序列不同而波动（最优和最差都是这个）。

  本质上就是：使用二分法和递归，将总数组的排序拆解为两个分数组排序，然后再将两个排好序的分数组进行合并。



算法流程：

> 设总函数为`merge_sort(nums, l, r)`，代表将nums数组中`l~r`段进行排序

1. 递归划分

计算数组中点`m`，递归地划分左子数组和右子数组：`merge_sort(nums, l, m)`、`merge_sort(nums, m+1, r)`。当函数传入的参数`l>=r`时，代表此时处理的子数组长度为1或0，此时终止划分，开始合并。（**此时`l~m`、`m+1~r`这两段内部都已经排好**）

2. 合并子数组

   （相当于合并两个有序数组任务）

   1. 将此次处理的子数组`nums[l:r+1]`暂存到辅助数组`tmp`中（注意由于是切片因此tmp和nums不共用一段内存，二者不会相互影响）
   2. 开始**循环合并**：本质上就是**合并两个有序数组**的问题，使用双指针分别在两个数组上移动，**每个时刻都挑较小的那个加入到原数组`nums`的相应位置**

      设置双指针`i,j`分别位于`tmp`中左/右段的首元素（注：`nums`中此次处理的子数组的左边界、中点、右边界分别为`l,m,r`，对应到`tmp`中分别为`0,m-l,r-l`），然后开始遍历整个`nums`数组（`k : 0~n`），每一步都看看`nums[k]`应该选用哪个指针指向的元素：

      1. 当`i==m-l+1`时，代表左段已经合并完（左指针越过了左段的右边界），因此添加右段元素`tmp[j]`（也即`nums[k]=tmp[j]`），并将右指针右移一位：`j=j+1`；
      2. 否则当`j==r-l+1`时，代表右段已合并完（右指针越过了右段的右边界），因此添加左段元素`tmp[i]`（也即`nums[k]=tmp[i]`），并将左指针右移一位：`i=i+1`；

      - 否则（两个指针都没越界）若`tmp[i]<=tmp[j]`，则添加左子数组元素`tmp[i]`（也即`nums[k]=tmp[i]`），并将左指针右移一位：`i=i+1`；
      - 否则（两个指针都没越界）若`tmp[j]<=tmp[i]`，则添加右子数组元素`tmp[j]`（也即`nums[k]=tmp[j]`），并将右指针右移一位：`j=j+1`；

```python
def merge_sort(nums, l, r):
    # 终止条件
    if l >= r: return
    # 递归划分数组
    m = (l + r) // 2
    merge_sort(nums, l, m)
    merge_sort(nums, m + 1, r)
    # 合并子数组
    tmp = nums[l:r + 1]        # 暂存需合并区间元素
    i, j = 0, m - l + 1        # 两指针分别指向左/右子数组的首个元素
    for k in range(l, r + 1): # 遍历合并左/右子数组
        if i == m - l + 1:
            nums[k] = tmp[j]
            j += 1
        elif j == r - l + 1:
            nums[k] = tmp[i]
            i += 1
        elif tmp[i] <= tmp[j]:
            nums[k] = tmp[i]
            i += 1
        elif tmp[j] <= tmp[i]:
            nums[k] = tmp[j]
            j += 1

# 调用
```

```
nums = [3, 4, 1, 5, 2, 1]
merge_sort(nums, 0, len(nums) - 1)
```

- **堆排序**

  对于从小到大进行排序，使用堆排序的方法为：

  - 首先，将整个树构造为一个**大顶堆**，构造堆的代码实现`def heapify`参考上文手动实现大顶堆
  - 然后，将堆顶元素（全局最大值）和最末端元素进行交换，此时最末端元素就成了最大值
  - 然后，将最末端元素排除在外，然后将剩下的元素重新调整为堆
  - 然后，将堆顶元素（当前最大值，全局第二大值）和倒数第二个末端元素进行交换，此时倒数第二个末端元素就是全局第二大值
  - 然后，将末尾的两个元素（全局第二大值和最大值）排除在外，然后将剩下的元素重新调整为堆
  - 如此不断循环，每次调整都使得堆顶为当前最大值，然后将这个堆顶扔到末尾的某个位置，使得末尾的几个元素是排好序的。最终即可**原地**将整个数组排好。

  最优、平均、最差时间复杂度都是$O(N\log N)$。空间复杂度为$O(1)$，这是它的优势。

  适用于：**要求空间复杂度为$O(1)$，或只想要获得前k个最大/最小元素的情况**

```
def heap_sort(nums):
    n = len(nums)

    # 构建最大堆（从最后一个非叶子节点开始）
    for i in range(n // 2 - 1, -1, -1):
        heapify(nums, n, i)

    # 逐个提取元素
    for i in range(n - 1, 0, -1):
        # 将当前最大值（堆顶）移到末尾
        nums[0], nums[i] = nums[i], nums[0]
        # 调整剩余元素使其满足堆性质
        heapify(nums, i, 0)

# 同前文实现，将无序列表转为堆
def heapify(nums, n, i):
    """
    调整以i为根的子树，使其满足最大堆性质
    n: 堆的大小
    i: 当前节点的索引
    """
    largest = i  # 假设当前节点是最大值
    left = 2 * i + 1  # 左子节点
    right = 2 * i + 2  # 右子节点

    # 如果左子节点存在且大于当前最大值
    if left < n and nums[left] > nums[largest]:
        largest = left
```

```
    # 如果右子节点存在且大于当前最大值
    if right < n and nums[right] > nums[largest]:
        largest = right

    # 如果最大值不是当前节点，交换并继续调整
    if largest != i:
        nums[i], nums[largest] = nums[largest], nums[i]
        heapify(nums, n, largest)
```

非比较型排序

上文的比较型排序依赖各个元素之间的比较，可以证明任何这种排序方法最坏情况下的时间复杂度都不低于$O(N\log N)$。如果想要突破这一限制，则需要使用非比较排序，最优能达到$O(N)$。

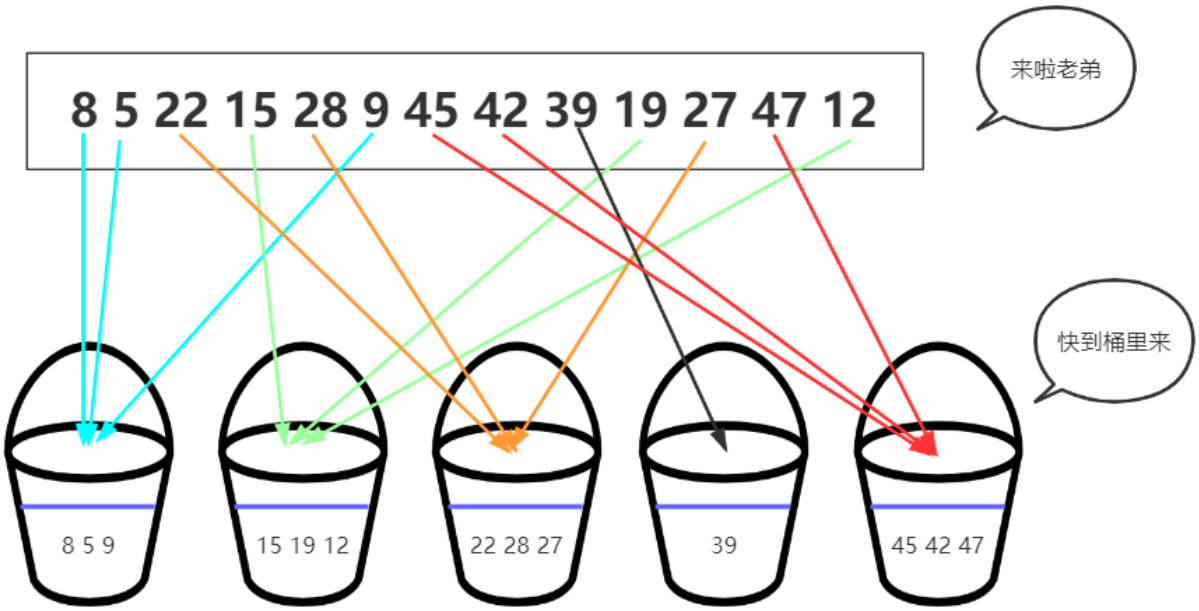
- **桶排序**

  思想为将数组分散到有限数量的桶里，每个桶内部再分别进行排序（桶内部排序可以使用其他排序算法，也可以递归地继续使用桶排序），最后再将每个桶的结果连接起来。其比较适用于**数组均匀地分布在某个范围内**的情况，可以使得每个桶中的元素数量比较均衡。**最优$O(N)$（元素分布足够均匀）**，最差$O(N\log N)$（元素全都聚集在一个桶内，退化为普通快速排序，实际上是$O(N+N\log N)$）

  > 如果考虑桶的数量k的话，则复杂度也可写成$O(N+k)$，其中k是遍历各个桶（并将它们各自排序）。因此桶的数量也不宜过大，至少应该不超过N个，这样才能保证线性复杂度。

  算法步骤：

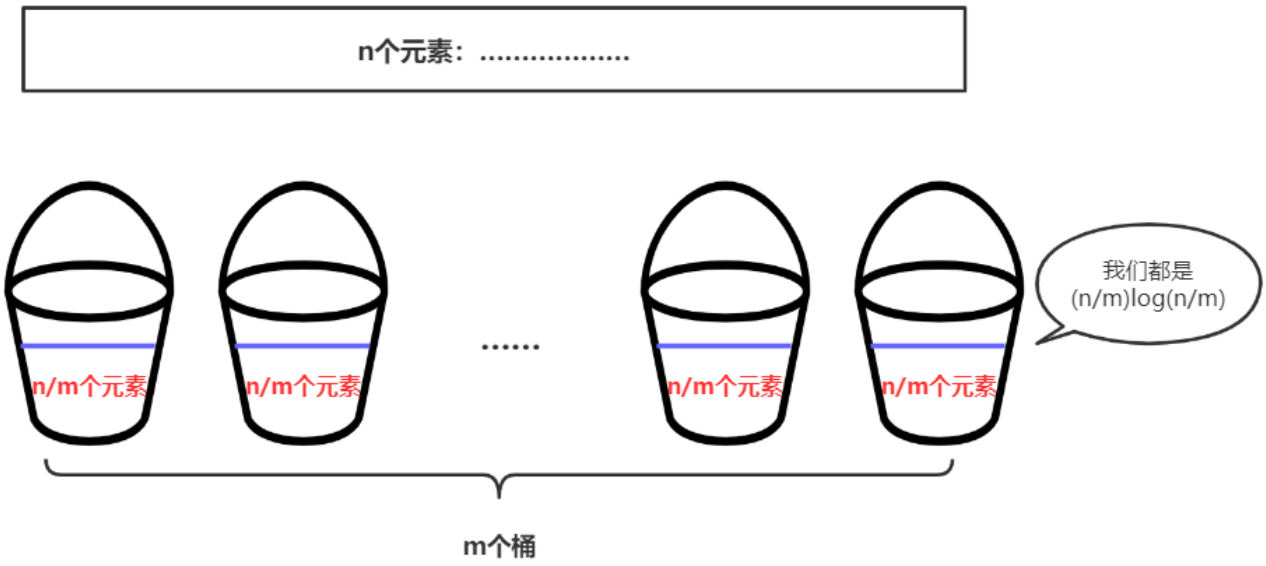
  - 预先设置好每个桶的容量，然后用数组的数值范围（max-min）除以桶容量，得到所需桶的数量。这样一来，每个桶负责保存某个数据范围内的元素。
  - 然后从头到尾遍历一遍数据，将每个数据根据它的数值大小放到它应该在的桶里。这一步的复杂度为$O(N)$。
  - 然后，依次对每个桶内部单独进行排序，这一步可以进一步递归使用桶排序，也可以使用快排等其他排序算法



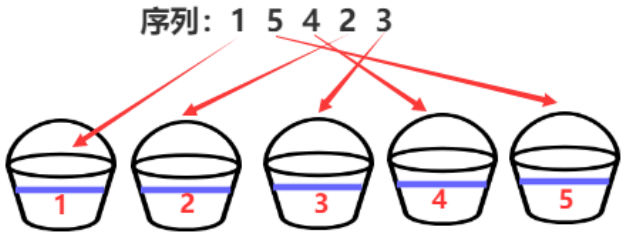
桶内再进行排序(排序方法自己选择)

根据桶和桶内顺序得到最终有序序列：5 8 9 12 15 19 22 27 28 39 42 45 47

复杂度：假设各个桶内元素数量比较均匀，设元素数量为N，桶数量为M，且每个桶内使用快排。则遍历数据放桶这一步的时间复杂度为$O(N)$，每个桶内N/M个元素快排的平均复杂度为$O((N/M)\log(N/M))$，M个桶就是$O(M(N/M)\log(N/M)) = O(N(\log N - \log M))$。最终的总复杂度为：$O(N + N(\log N - \log M))$。



可见，当桶数量就等于元素数量且数据分布绝对均匀时，有$N = M$，此时相当于每个桶内不再需要单独排序，**最优**总复杂度就是分桶时产生的$O(N)$



如果数据分布特别不均匀，绝大部分元素都聚集在了一个桶里，则分桶就没什么意义了，相当于这个桶内进行了几乎所有元素的快排，**最差**总复杂度变为$O(N + N\log N) \to O(N\log N)$

```
def bucket_sort(nums, bucket_size=5):
    # 首先找到最小值和最大值，确定数据范围
    min_val, max_val = min(nums), max(nums)

    # 计算桶数，为数据范围除以桶大小再+1
    bucket_count = (max_val - min_val) // bucket_size + 1
    # 创建各个桶，每个桶为一个列表
    buckets = [[] for _ in range(bucket_count)]

    # 然后从头到尾遍历一遍数组并将各个元素放入它应该在的桶里
    for num in nums:
        bucket_idx = (num-min_val) // bucket_size
        buckets[index].append(num)

    # 最后，分别对每个桶进行独立地排序，并将各个桶的排序结果连接起来作为最终排序结果
```

```
    sorted_nums = []
    for bucket in buckets:
        sorted_nums.extend(sorted(bucket))

    return sorted_nums
```

- **计数排序：**

  适用于**整数排序**，通过统计数组中每个整数元素出现的次数，进一步通过累加得知每个元素前边有多少个元素（小于等于它的元素），这样即可将各个元素插入其在排好的数组中所在的位置。最优、最坏、平均时间复杂度都是$O(N+k)$，**当数据范围k不是很大时比较适用**，复杂度接近$O(N)$，若数据范围k过大则可能导致复杂度上升。

  流程：

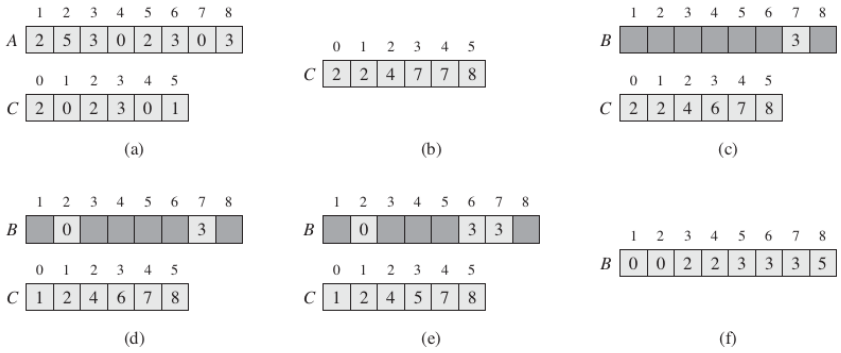
  - 找出数组中的最小值`min_val`和最大值`max_val`，然后设置一个大小为`max_val-min_val+1`的辅助数组`count`，其中`count[i]`表示值等于`i-min_val`的元素的数量
  - 然后，遍历一遍数组，统计各个元素出现次数
  - 然后，将数组从前往后进行累加，使得`count[i]`变为值小于等于`i-min_val`的元素的数量
  - 然后设置空的结果数组`result`，其形状和原始数组相同，用于插入原始数组中的各个元素。然后即可遍历原始数组，每遍历到一个元素就将其插入到`result`的相应位置，并将其在`result`中的计数值-1（因为可能这个数出现了多次，这里用掉了一次，还剩下`count[i-min_val]-1`个元素小于等于它）

    这里通常选择从后往前遍历原始数组，可以确保排序稳定性。从前往后排序结果也是正确的，但有可能破坏排序稳定性。

```
def counting_sort(nums):
    """
    计数排序 - 支持正负数
    """
    if len(nums) <= 1:
        return nums

    # 找出最大值和最小值
    min_val = min(nums)
    max_val = max(nums)

    # 创建计数数组
    count_size = max_val - min_val + 1
    count = [0] * count_size

    # 统计每个元素出现的次数
    for num in nums:
        count[num - min_val] += 1

    # 计算累计次数
    for i in range(1, count_size):
        count[i] += count[i-1]

    # 构建结果数组（从后往前遍历保持稳定性）
```

```
    result = [0] * len(nums)
    for i in range(len(nums)-1, -1, -1):
        num = nums[i]
        pos = count[num - min_val] - 1
        result[pos] = num
        count[num - min_val] -= 1

    return result
```

示例：A为原始数组，C为count数组，B为结果数组



- **基数排序：**

  适用于**整数排序**，其从最低位到最高位按照数字的每一位进行排序。对于n个k进制d位数，假设每一位的排序都是用计数排序，则d位数的总排序用时为$O(d(n+k)) \to O(n)$。适用于**数字位数不多但数值范围很大时。**

  算法流程：

  - 首先找到数组中最大的数，然后确定它的位数
  - 然后从**最低位**开始，从低到高对每一位进行计数排序，直到排到最高位（注意这里和直觉相反，并不是先比最高位，而是先比最低位）

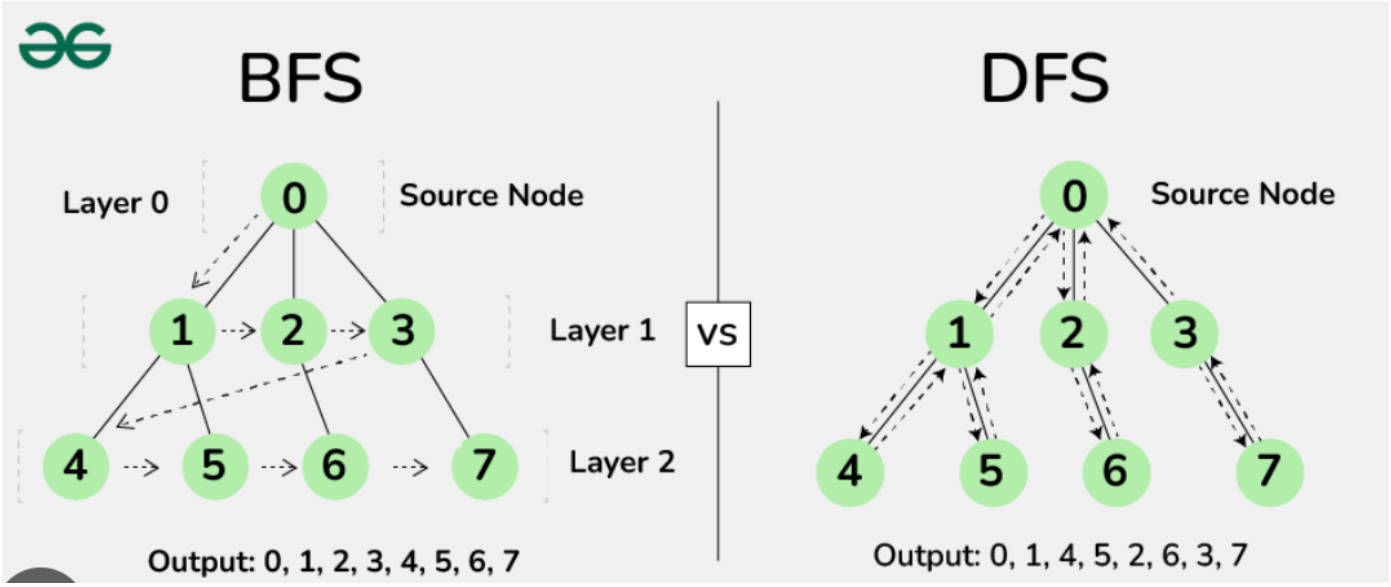
  例如，对于`[329, 457, 839, 355]`，首先针对最后一位进行排序（若最后一位相等则保持原先相对顺序不变，从而确保稳定排序），然后对倒数第二位进行排序，最后对倒数第三位排序：

  ```
  355        329        329
  457        839        355
  329        355        457
  839        457        839
  ```

```
def radix_sort(nums):
    """
    基数排序 - 支持正整数
    """
    if len(nums) <= 1:
        return nums

    # 找到最大数确定位数
    max_num = max(nums)
    exp = 1  # 从个位开始
```

```
    # 使用计数排序对每一位进行排序
    while max_num // exp > 0:
        counting_sort_by_digit(nums, exp)
        exp *= 10

    return nums

def counting_sort_by_digit(nums, exp):
    """
    对特定数位进行计数排序
    """
    n = len(nums)
    output = [0] * n
    count = [0] * 10  # 0-9的数字

    # 统计当前数位上每个数字的出现次数
    for i in range(n):
        digit = (nums[i] // exp) % 10
        count[digit] += 1

    # 计算累计次数
    for i in range(1, 10):
        count[i] += count[i-1]

    # 构建输出数组（从后往前保持稳定性）
    for i in range(n-1, -1, -1):
        digit = (nums[i] // exp) % 10
        output[count[digit] - 1] = nums[i]
        count[digit] -= 1

    # 将结果复制回原数组
    for i in range(n):
        nums[i] = output[i]
```

搜索算法



BFS

广度优先搜索（BFS）

使用先进先出的**队列**来实现。

步骤：

1. 把起始点放入queue
2. 重复以下步骤，直到queue空为止：
    1. 从queue中取出队首（并将其结果保存到访问队列中）
    2. 找出与取出的这个队首邻接且尚未遍历的点，进行标记（说明已被遍历到），然后将它们依次入队

    注意：每个点实际上会被经过两次，第一次是“遍历到”，会使得其被标记并被加入队列；第二次是“访问到”，会使得其出队，并返回该点数据（保存到返回结果列表中或打印出来），对该点的访问彻底结束。

应用场景：

- BFS的两个场景：**层序遍历**、**最短路径**
- 找距离某一点的最短路，但路径不唯一，最先搜索到满足条件的就是最短的路径
- **大范围**的查找
- 出现 “**最短**”、“**最少**”类似字眼的可以优先考虑

图遍历的示例：

```
# 创建一个字典，用于存储图。字典相当于映射关系，通过键值对进行读取。适用于图只有少量的点，数据过多使用python类更为合适
graph = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D", "E"],
    "D": ["B", "C", "E", "F"],
    "E": ["C", "D"],
    "F": ["D"]
}
```

```
# 开始BFS遍历
# graph是图数据，s是图的起点
def BFS(graph, s):
    # 创建一个队列，用于存储未访问过的点
    from collections import deque
    queue = deque()
    # 放入起点
    queue.append(s)
    # 创建一个集合，用于存放已遍历过的点，例如起点s
    seen = set()
    seen.add(s)
    # 循环读queue
    while queue:
        # 队首出队，对该点访问完成
        vertex = queue.popleft()
        # 读取该点相邻的点
        nodes = graph[vertex]
        # 判重：循环判断相邻的点是否读过
        for w in nodes:
            if w not in seen:
                # 若某个相邻点没有被遍历过，则将其入队并标记为遍历过
                queue.append(w)
                seen.add(w)
        # 输出遍历节点
        print(vertex, end= ' ')
```

上例中是使用了一个`set()`作为判断是否遍历过的依据。

Eg1：网格中的最短路径

给你一个 `m * n` 的网格，其中每个单元格不是 `0`（空）就是 `1`（障碍物）。每一步，您都可以在空白单元格中上、下、左、右移动。

如果您 **最多** 可以消除 `k` 个障碍物，请找出从左上角 `(0, 0)` 到右下角 `(m-1, n-1)` 的最短路径，并返回通过该路径所需的步数。如果找不到这样的路径，则返回 `-1` 。

解：乍一看似乎可以使用动态规划，但注意到由于每个点都可能来自于4个方向，而动态规划需要确保当前状态只依赖于之前计算过的状态，因此这个问题不应使用动态规划。而BFS由于是“逐层扩散”，因此也常用于寻找最短路径长度。

因此，本题思路为使用BFS进行逐层扩散搜索。由于遇到障碍物时有`k`次机会进行消除，因此每个入队的点不仅包含本身的坐标，还额外包含一个状态值`rest`，表示扩散到当前层时还有多少消除障碍的余量，初始值为`k`。

```
class Solution:
    def shortestPath(self, grid: List[List[int]], k: int) -> int:

        # 边界条件：总共只有一个点
        if len(grid) == 1 and len(grid[0]) == 1:
```

```
            return 0

        from collections import deque
        queue = deque()
        seen = set()

        # 每个点包含自己的坐标，以及到这个点时剩余的可以消除的机会（状态信息）
        start = (0,0,k)
        queue.append(start)
        seen.add(start)

        # 统计遍历到“第几层”了，也即到“当前层”的最短步数
        step = 0
        while queue:
            # 每遍历到一个新层，则step+1
            step += 1
            for _ in range(len(queue)):
                x0, y0, rest = queue.popleft()
                for x, y in [(x0-1,y0), (x0+1,y0), (x0, y0-1), (x0, y0+1)]:
                    if 0 <= x and x < len(grid) and 0 <= y and y < len(grid[0]):
                        # 若该点本身不是障碍物
                        if grid[x][y] == 0 and (x, y, rest) not in seen:
                            queue.append((x, y, rest))
                            seen.add((x,y,rest))
                            # 在加入队列时即可判断其是不是终点，若是则可结束遍历，不需要等到出队时再判断。（这也是step+=1）位于每轮循环开始时的道理
                            if x == len(grid)-1 and y == len(grid[0])-1:
                                return step
                        # 若该点本身是障碍物，且还有消除障碍物的余量，则也可以将其入队
                        elif grid[x][y] == 1 and rest > 0 and (x,y,rest-1) not in seen:
                            queue.append((x,y,rest-1))
                            seen.add((x,y,rest-1))
                            if x == len(grid)-1 and y == len(grid[0])-1:
                                return step
        return -1
```

Eg2：课程表（拓扑排序）

你这个学期必须选修 `numCourses` 门课程，记为 `0` 到 `numCourses - 1` 。

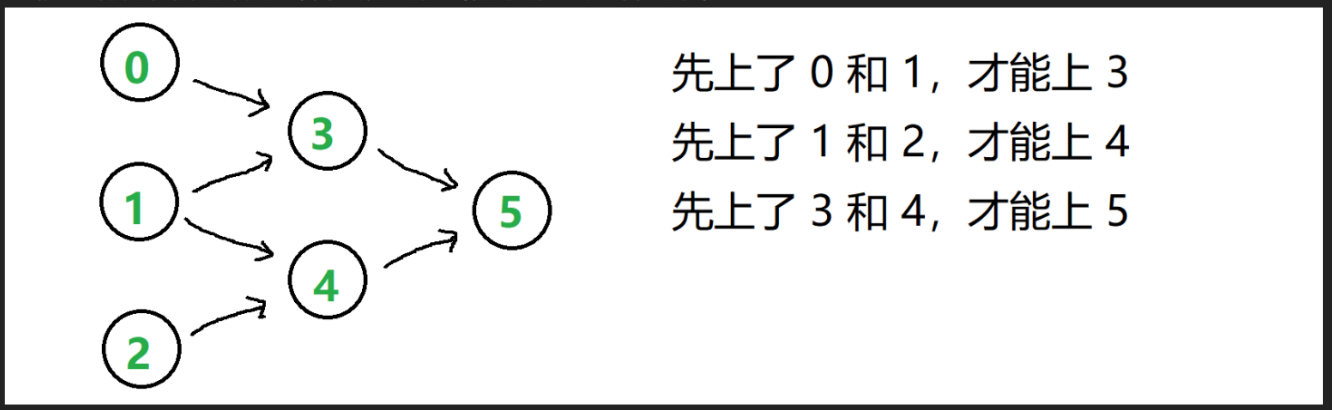
在选修某些课程之前需要一些先修课程。 先修课程按数组 `prerequisites` 给出，其中 `prerequisites[i] = [ai, bi]` ，表示如果要学习课程 `ai` 则 **必须** 先学习课程 `bi` 。

- 例如，先修课程对 `[0, 1]` 表示：想要学习课程 `0` ，你需要先完成课程 `1` 。

请你判断是否可能完成所有课程的学习？如果可以，返回 `true` ；否则，返回 `false` 。

解：可以将课程之间的依赖关系构建为有向图：

- 示例：`n = 6`，先决条件表：`[[3, 0], [3, 1], [4, 1], [4, 2], [5, 3], [5, 4]]`
- 课 `0, 1, 2` 没有先修课，可以直接选。其余的课，都有两门先修课。
- 我们用有向图来展现这种依赖关系（做事情的先后关系）：



- 这种叫 **有向无环图**，把一个 有向无环图 转成 线性的排序 就叫 **拓扑排序**。
- 有向图有 **入度** 和 **出度** 的概念：
  - 如果存在一条有向边 A --> B，则这条边给 A 增加了 1 个出度，给 B 增加了 1 个入度。
- 所以，顶点 0、1、2 的入度为 0。顶点 3、4、5 的入度为 2。

则只有当前时刻入度为0的节点（课程）可以上，每上完一门课则可以将其所有的直接后继课的入度都减1（也即它们的先修课减少了1门）

设置一个列表`in_degree`来保存各个节点当前的入度，一个字典`adj_dict`来保存各个节点的后继节点。使用BFS来对图进行遍历与消去：队列queue中保存的为当前入度为0的节点（初始时即为没有任何先修课的课程），每遍历过队列中的1个节点后，通过`adj_dict`获得其所有后继节点，并将各个后继节点的入度都-1，并检查此时是否出现了新的入度减到0的节点，并将其加入到队列中。

设置一个`num_courses_left`来统计当前还剩多少课没上完，每遍历过一个节点就将其减1。如果队列为空（此时不存在入度为0的节点）时发现`num_courses_left=0`，则说明都上完了，返回True，否则返回False（此时不存在入度为0的节点，但还有课程没上）

```
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        # 入度统计，in_degree[i]表示课程i的入度，也即其需要多少门直接先修课
        in_degree = [0 for _ in range(numCourses)]
        # 邻接关系字典，键为某门课，值为一个列表，保存了以该门课为直接先修课的所有课
        adj_dict = dict()

        # 遍历prereq数组，构建入度统计列表和邻接关系字典
        for course, pre_course in prerequisites:
            in_degree[course] += 1  # course多了一个需要先修的课
            if not adj_dict.get(pre_course):    # 第一次发现pre_course的后继课
                adj_dict[pre_course] = [course]
            else:
                adj_dict[pre_course].append(course)

        # 队列queue中存储的为当前入度为0的课程，也即当前不存在先修课可以直接上的课程
        from collections import deque
```

```
        queue = deque()
        # 将所有入度为0的课加入初始队列
        for i in range(len(in_degree)):
            if in_degree[i] == 0:
                queue.append(i)
        # 开始bfs
        num_courses_left = numCourses   # 当前剩余还未完成的课数
        while queue:
            course = queue.popleft()
            num_courses_left -= 1   # 完成一门课
            if adj_dict.get(course):    # 如果该课存在后继课，则可以将它们的入度都-1
                for sub_course in adj_dict[course]:
                    in_degree[sub_course] -= 1
                    if in_degree[sub_course] == 0:  # 如果某门课的入度此时减到了0，则它可以直接被上了，将其加入队列
                        queue.append(sub_course)

        return True if num_courses_left == 0 else False
```

Eg3：除法求值（有向有权重图求给顶点之间路径）

给你一个变量对数组 `equations` 和一个实数值数组 `values` 作为已知条件，其中 `equations[i] = [Ai, Bi]` 和 `values[i]` 共同表示等式 `Ai / Bi = values[i]` 。每个 `Ai` 或 `Bi` 是一个表示单个变量的字符串。

另有一些以数组 `queries` 表示的问题，其中 `queries[j] = [Cj, Dj]` 表示第 `j` 个问题，请你根据已知条件找出 `Cj / Dj = ?` 的结果作为答案。

返回 **所有问题的答案** 。如果存在某个无法确定的答案，则用 `-1.0` 替代这个答案。如果问题中出现了给定的已知条件中没有出现的字符串，也需要用 `-1.0` 替代这个答案。

注意：输入总是有效的。你可以假设除法运算中不会出现除数为 0 的情况，且不存在任何矛盾的结果。

注意：未在等式列表中出现的变量是未定义的，因此无法确定它们的答案。

示例：

```
输入：equations = [["a","b"],["b","c"]], values = [2.0,3.0], queries = [["a","c"],["b","a"],["a","e"],["a","a"],["x","x"]]
输出：[6.00000,0.50000,-1.00000,1.00000,-1.00000]
解释：
条件：a / b = 2.0, b / c = 3.0
问题：a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?
结果：[6.0, 0.5, -1.0, 1.0, -1.0 ]
注意：x 是未定义的 => -1.0
```

解：

本题可以通过建图+bfs进行解决。可以将变量作为节点，两个变量之间的商作为二者节点之间的边的权重，例如对于`[A, B] = 6.0`，则`weight_{A->B}=6.0`，`weight_{B->A}=1.0/6.0`。这样一来，使用`equations`建好图之后，对于`queries`中的每对变量，如果能找到二者之间的路径则可以算出它们的除法结果，如果二者不全在图中或找不到连通的路径，那么就无法算出二者之间的除法结果。可以使用bfs来从给定起始点出发，寻找到达目标点的路径

具体而言，使用`graph=defaultdict(dict)`来构建二阶字典作为图，`graph[A][B]`即为A->B的除法结果。从初始点开始，在bfs过程中，一旦遍历到目标点则说明找到了路径，返回累积乘法结果即可。

> 二阶字典`graph`形如：

```
graph = {
'A': {'B': 2, 'C': 4, ...},
'B': {'A': 0.5, ...},
...
}
```

```python
class Solution:
    def calcEquation(self, equations: List[List[str]], values: List[float], queries: List[List[str]]) -> List[float]:
        from collections import deque, defaultdict
        # 构建有向有权重图
        graph = defaultdict(dict)
        # 遍历equations，构建已知图
        for (A, B), value in zip(equations, values):
            graph[A][B] = value
            graph[B][A] = 1.0 / value

        # 定义bfs函数，其试图找到两点之间的路径，并返回两点之间的累积倍数
        def bfs(start, end):
            # 如果任意一点不在图中，则二者之间肯定无路径
            if not start in graph or not end in graph:
                return -1.0
            queue = deque()
            visited = set()
            queue.append((start, 1.0))  # 队列中每个元素记录从start到当前点的累积倍数
            visited.add(start)
            while queue:
                node, cur_product = queue.popleft()
                # 如果此时已经遍历到了end点，则返回此时点累积倍数作为结果
                if node == end:
                    return cur_product
                # 将相邻点入队，graph[node]本身也是一个dict，每对key:value为一个相邻点和边权重
                for neighbor, weight in graph[node].items():
                    if neighbor not in visited:
                        # 入队，并更新start到该点到累积倍数
                        queue.append((neighbor, cur_product * weight))
                        visited.add(neighbor)
            # 没有找到能从start到end的路径
            return -1.0
```

```
        # 针对queries中的每对start-end节点，进行bfs并统计累积倍数
        result = []
        for C, D in queries:
            result.append(bfs(C,D))
        return result
```

Eg4. 删除无效的括号

给你一个由若干括号和字母组成的字符串 `s` ，删除最小数量的无效括号，使得输入的字符串有效。

返回所有可能的结果。答案可以按 任意顺序 返回。

示例 1：

```
输入：s = "()())()"
输出：["(())()","()()()"]
```

示例 2：

```
输入：s = "(a)())()"
输出：["(a())()","(a)()()"]
```

解：

对于此类“最少操作达到某个目标”问题，可以将其使用BFS解决。本题中，将一个子串视为一个节点，则其各个“相邻后继节点”就是它去掉一个括号后的所有可能的子串。每去掉1个括号，则可将其视为搜索更深的1层，当在某一层中首次发现合法子串时，则该层就是删除最小数量括号能达到的子串的集合，因此可以继续收集该层的所有合法子串。当该层所有合法子串收集完毕后，就无需再继续删除括号搜索更深层了，因为本层已经是删除最小数量括号后能获得合法子串的集合了，将此时收集到的所有结果返回即为最终答案。

```
class Solution:
    def removeInvalidParentheses(self, s: str) -> List[str]:
        # 该函数用于检验一个给定的含有括号和字母的字符串是否有效
        def is_valid_str(s):
            cnt = 0     # 每遇到一个'('则+1，遇到一个')'则-1
            for char in s:
                if char == '(':
                    cnt += 1
                if char == ')':
                    cnt -= 1

                if cnt < 0: # 如果某时刻')'数量大于了其左侧'('的数量，则一定无效
                    return False
            return True if cnt == 0 else False  # 最终如果'('和')'都消完了则是有效的

        # 准备bfs
        from collections import deque
        queue = deque()
        visited = set()
```

```
        queue.append(s)
        visited.add(s)

        found = False   # 此时是否已经找到合法子串，如果已经找到的话那就无需再继续构建下一层（也即无需把再删除一个括号后的子串状态入队），只需将当前层遍历完即可
        results = []

        # 开始bfs
        while queue:
            # 遍历一层（也即删去某个特定数量括号后可能达到的所有子串）
            for _ in range(len(queue)):
                cur_str = queue.popleft()
                # 如果当前子串有效，则说明此时找到了有效串，将其加入结果中
                if is_valid_str(cur_str):
                    results.append(cur_str)
                    found = True

                # 如果本层已经找到了，则不用再继续构建下一层了，把当前层遍历完（从而搜集完所有有效子串）即可
                if found:
                    continue

                # 继续构建下一层（也即把继续删除一个括号后的子串状态入队）
                # 遍历当前串删除每个括号后的子串，并将它们入队
                for i in range(len(cur_str)):
                    if cur_str[i] == '(' or cur_str[i] == ')':
                        sub_str = cur_str[:i] + cur_str[i+1:]   # 删去cur_str[i]位置括号后的子串
                        if sub_str not in visited:
                            queue.append(sub_str)
                            visited.add(sub_str)

            # 遍历完一层后，如果发现当前层中已经找到合法子串，则返回当前层收集的所有合法子串结果即可，它们就是删除最小数量无效括号后的所有可能子串，无需再继续删括号构建下一层
            if found:
                return results

        return results if results else [""]
```

DFS

深度优先搜索（DFS）

对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问**一次**。当节点v的所在边都已被探寻过或者在搜寻时结点不满足条件，搜索将回溯到发现节点v的那条边的起始节点。整个进程反复进行直到所有节点都被访问为止。

使用先进后出的**栈**来实现。和上述BFS的区别几乎仅在于使用**栈**而不是队列来保存已被遍历过而待访问的节点。DFS的访问顺序不唯一，取决于对于每个被访问点的相邻点的压栈顺序

遍历图的示例：

```
# 创建一个字典，用于存储图。字典相当于映射关系，通过键值对进行读取。适用于图只有少量的点，数据过多使用python类更为合适
graph = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D", "E"],
    "D": ["B", "C", "E", "F"],
    "E": ["C", "D"],
    "F": ["D"]
}

# 开始DFS遍历
# graph是图数据，s是图的起点
def DFS(graph, s):
    # 创建一个数组作为栈，用于存储未访问过的点
    stack = []
    # 放入起点
    stack.append(s)
    # 创建一个集合，用于存放已遍历到的点，例如起点s
    seen = set()
    seen.add(s)
    # 循环读stack
    while (len(stack) > 0):
        # 通过queue.pop()读取栈最后一个
        vertex = stack.pop()
        # 读取每个点相邻的点
        nodes = graph[vertex]
        # 判重：循环判断相邻的点是否遍历过
        for w in nodes:
            if w not in seen:
                stack.append(w)
                seen.add(w)
        # 输出遍历节点
        print(vertex, end= ' ')
```

- 选择策略：
    - BFS是用来搜索最短径路的解是比较合适的，比如求最少步数的解，最少交换次数的解，因为BFS搜索过程中遇到的解一定是离根最近的，所以遇到一个解，一定就是最优解，此时搜索算法可以终止。这个时候不适宜使用DFS，因为DFS搜索到的解不一定是离根最近的，只有全局搜索完毕，才能从所有解中找出离根的最近的解。（当然这个DFS的不足，可以使用迭代加深搜索ID-DFS去弥补）。
    - 空间优劣上，DFS是有优势的，DFS不需要保存搜索过程中的状态，而BFS在搜索过程中需要保存搜索过的状态，而且一般情况需要一个队列来记录。
    - DFS适合搜索全部的解，因为要搜索全部的解，那么BFS搜索过程中，遇到离根最近的解，并没有什么用，也必须遍历完整棵搜索树，DFS搜索也会搜索全部，但是相比DFS不用记录过多信息，所以搜素全部解的问题，DFS显然更加合适。

Eg1：岛屿数量

给你一个由 `'1'`（陆地）和 `'0'`（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

解：

本题实际上用bfs和dfs都行，只需要完成遍历功能即可，不需要用到bfs或dfs的特定性质。

遍历grid中的每一个点，如果它是陆地`"1"`，则以它为根进行bfs或dfs，从而遍历和它相连的所有陆地，且每遍历到一个陆地点都将其设为`"0"`（相当于每个岛屿都用bfs或dfs来将其干掉）。统计遍历整个grid中进行的bfs或dfs次数，其就是岛屿数量。

```
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        nr, nc = len(grid), len(grid[0])
        # dfs, 将当前节点所在岛屿的所有1设为0
        def dfs(grid, r, c):
            stack = [(r,c)]
            seen = set()
            seen.add((r,c))
            while stack:
                r0, c0 = stack.pop()
                grid[r0][c0] = '0'
                for x,y in [(r0-1,c0),(r0+1,c0),(r0,c0-1),(r0,c0+1)]:
                    if 0 <= x < nr and 0 <= y < nc:
                        if grid[x][y] == '1' and (x,y) not in seen:
                            stack.append((x,y))
                            seen.add((x,y))
        cnt = 0
        for r in range(nr):
            for c in range(nc):
                if grid[r][c] == '1':
                    # 每进行一次dfs，则记录一个岛屿
                    cnt += 1
                    dfs(grid, r, c)
        return cnt
```

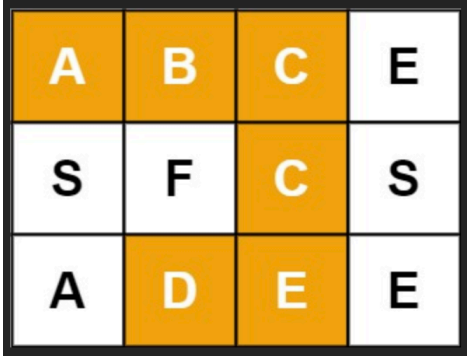
当然是事实也可以直接在入队的时候就置0，这样可以省掉`seen`带来的开销

Eg2. 单词搜索

给定一个 `m x n` 二维字符网格 `board` 和一个字符串单词 `word` 。如果 `word` 存在于网格中，返回 `true` ；否则，返回 `false` 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

示例 1：




```
输入：board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],
word = "ABCCED"
输出：true
```

解：

本题是典型的**“DFS+回溯”**问题：想找一个符合某种条件的**路径**，由于DFS天然适合形成路径因此采用DFS来进行路径构建：构建过程中，每遍历到一个节点时就查看它四周的邻居是否符合条件，并将这些未来有可能用于组成有效路径的节点入栈备用。构建过程中需要不断尝试，每当发现添加的一个节点走进了死胡同时就将它撤销（这一步通常不需要手动处理，stack弹出其实就起到了撤销回溯的效果），然后翻回头尝试stack中保存的其他可能路径。

注意：这种路径构造问题有时也需要考虑节点访问判重（比如同一个路径中不能重复经过某一个节点），此时最好不要用传统dfs中的visited集合来维护已访问过的节点，它会在回溯撤销时带来很大麻烦（例如，有时撤销一个节点后，它所在的路径上的一大堆节点可能都跟着没了，但如果只从visited集合中删去当前撤销的节点的话，前边那些跟着撤销的节点的状态仍无法恢复成un-visited，导致其他路径也不能用它们）。最好的办法是：**在每个节点入栈时，将首节点到当前节点的路径也跟着入栈**，也即栈中的每个节点不仅维护其坐标等信息，还会有一个它专属的迄今为止的路径状态信息。

在本题中，首先是逐个遍历键盘上的各个节点，一旦发现某个节点和`word[0]`吻合，就以它为首节点开始尝试构造一条有效路径。在stack中，每个节点维护的状态包括其自己的坐标、其对应了`word`中的第几个字母、从首节点到当前节点的路径集合（用于判重，因此用一个集合来记录迄今为止的路径信息即可，不需要用列表记录路径前后顺序信息）。每遍历到一个节点时，设它本身对应`word[k]`，分别查看一下它上下左右的4个邻居是否有能和`word[k+1]`对应（且当前路径中未访问过），对于满足条件的有效节点，将其加入栈中以备以后尝试。当遍历到某个节点时，发现它对应了`word[n-1]`，则说明首节点到这个节点的路径就是一个完整的有效路径，可以返回True。

```python
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        if not board or not word:
            return False

        rows, cols = len(board), len(board[0])

        # 遍历网格中的每一个字母
        for i in range(rows):
            for j in range(cols):
                # 如果发现某个字母和word的首字母吻合，则从它开始，进行一波dfs尝试
                if board[i][j] == word[0]:
```

```
                    stack = []
                    # stack中保存的是：
                    # 可以用于构建序列的一个元素的坐标(i,j)
                    # 它对应word中的第几个字母（一开始的首字母对应word[0]，所以是0）
                    # 从本轮首字母节点到当前节点的路径上，已经使用过的所有节点构成的集合
                    path_visited = set()
                    path_visited.add((i, j))
                    stack.append((i, j, 0, path_visited))

                    while stack:
                        # 当前弹出的节点board[x][y]和word[k]是吻合的
                        x, y, k, path_visited = stack.pop()
                        # 如果发现此时弹出的元素正好对应word中最后一个元素，则说明完成了序列构建，返回
True
                        if k == len(word) - 1:
                            return True

                        # 尝试当前节点的4个方向的相邻节点，看看有没有和word[k+1]吻合的
                        for nx, ny in [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
                            if 0 <= nx < rows and 0 <= ny < cols:
                                if (nx, ny) not in path_visited and board[nx][ny] ==
word[k + 1]:
                                    # 如果发现某个相邻节点和word[k+1]吻合，则将其入栈，以备以后使
用
                                    path_visited_sub = path_visited.copy()
                                    path_visited_sub.add((nx, ny))
                                    stack.append((nx, ny, k + 1, path_visited_sub))

        return False
```

有权图最小路径问题（Dijkstra）

对于一个有权图，想要找到某个源节点距离其他各个节点的最小路径值，可使用Dijkstra算法。

算法流程：

- 初始化：
  - 创建一个距离字典`distances`，keys为图中各个节点，values为各个节点距离源节点的当前最小路径值，用于存储各个节点距离源节点的当前最小路径值。源节点本身的值初始化为0，其他节点的值初始化为`inf`。当算法流程结束时，其中的各个值即为各个节点距离源节点的真正最小路径值。
  - 创建一个最小堆（优先队列）`heap`，用于动态地存储那些有潜力为其邻居带来更小距离的节点，堆中的元素为`(当前最小距离, 节点)`。初始化时放入`(0, start)`。
- 主循环：
  - 每次从优先队列中取出当前距离源节点路径最小的节点，以及其对应的当前距离
  - 如果发现当前距离大于`distances`中记录的该节点的已知最小距离，则跳过，继续遍历队列（因为这说明这个距离入队后，处理其他节点时又找到了比它更小的距离，也即这个距离已经过时了，不必继续用它来更新它的邻居节点的最小距离）

- 若未跳过，则开始遍历当前节点的各个邻居：
  - 对于每个邻居，算出其经由当前节点节点到达源节点的最小路径：`当前距离（当前节点到源节点的最短距离）+ 当前节点和该邻居之间的路径权重`
  - 如果这个路径小于`distances`中记录的这个邻居的已知最小路径，则说明找到了该邻居的一个更小距离，将其记录在`distances`中并放入优先队列（因为它有可能进一步给它的邻居们带来更小的距离）
- 当优先队列为空时，说明已经处理好了所有可达节点，此时算法流程结束

```
import heapq

def dijkstra(graph, start):
    # 初始化距离字典，所有节点距离设为无穷大
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0

    # 优先队列，存储(距离, 节点)元组
    heap = [(0, start)]

    while heap:
        current_dist, current_node = heapq.heappop(heap)

        # 如果当前距离大于已知距离，跳过
        if current_dist > distances[current_node]:
            continue

        # 遍历邻居
        for neighbor, weight in graph[current_node].items():
            distance = current_dist + weight

            # 如果找到更短路径，则更新
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))

    return distances
```

链表

链表：

```
# 普通单向链表
class ListNode:
    def __init__(self, x):
        self.val = x     # 节点值
        self.next = None # 后继节点引用

# 双向链表
class BiListNode:
    def __init__(self, x):
        self.val = x     # 节点值
        self.next = None # 后继节点引用
        self.prev = None # 前序节点引用
```

- eg1：翻转链表

  只需将每个节点的指针指向其前序节点即可。注意需要一直用`pre`来保存前序节点，且用`tmp`保存后序节点（避免把指针指向前序节点后找不到后续节点）

  ```
  # Definition for singly-linked list.
  # class ListNode:
  #     def __init__(self, x):
  #         self.val = x
  #         self.next = None

  class Solution:
      def reverseList(self, head: ListNode) -> ListNode:
          cur, pre = head, None
          while cur:
              tmp = cur.next
              cur.next = pre
              pre = cur
              cur = tmp
          return pre
  ```

- eg2：环形链表：

  给定一个链表，判定是否有环。

  解：

  使用**快慢指针法**，二者均从head开始，快指针一次走两格，慢指针一次走一格，若二者在某一时刻相遇，则说明有环；若快指针在某一时刻到达None（链表尾），则说明无环。

  注意边界条件：一方面对于链表本身的判定，需要分别判定`head is None`和`head.next is None`两种情况；且在每轮让`fast`指针移动第二步之前，也要判断一下其在走完第一步后是否到达了None。

  ```
  # Definition for singly-linked list.
  # class ListNode(object):
  #     def __init__(self, x):
  #         self.val = x
  #         self.next = None

  class Solution(object):
  ```

```
    def hasCycle(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        if not head: return head
        if not head.next: return False
        slow = fast = head
        while fast:
            slow = slow.next
            fast = fast.next
            if fast:
                fast = fast.next
            if fast == slow:
                return True
        return False
```

- eg3：合并两个有序链表

  将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。例如：

  ```
  输入：l1 = [1,2,4], l2 = [1,3,4]
  输出：[1,1,2,3,4,4]
  ```

  解：可以借鉴归并排序中对两个有序数组进行合并的方法，更好理解一些。设置一个新的链表（伪链表头`dummyHead`）作为排序后的总链表，然后设置三个指针`temp, temp1, temp2`分别指向总链表、链表1、链表2的相应位置，然后分别移动这三个指针，每次都比较`temp1`和`temp2`指针分别指向的元素，并将较小地放入总链表中：

  ```
  def merge(head1, head2):
            # 排序后的总链表的伪表头
            dummyHead = ListNode()
            # 三个指针，分别指向排序后的总链表、链表1、链表2的相应位置
            temp, temp1, temp2 = dummyHead, head1, head2
            # 开始移动三个指针
            while temp1 and temp2:
                if temp1.val <= temp2.val:
                    temp.next = temp1
                    temp1 = temp1.next
                else:
                    temp.next = temp2
                    temp2 = temp2.next
                temp = temp.next
            # 此时temp1或temp2移到了终点，则将没走到中点的那一个链表剩余的部分直接接到总链表后边
            if temp1:
                temp.next = temp1
            elif temp2:
  ```

```
                temp.next = temp2
            return dummyHead.next
```

- eg4：排序链表

  使用**归并排序**，先用**快慢指针法**（快指针一次走两格，慢指针一次走一格，当快指针走到头时慢指针恰好在中间，注意和判圈法的双指针一样，也要将`head is None`和`head.next is None`两种情况单拿出来判断）找到中点节点。然后使用归并排序的二分递归+合并，即可完成排序。

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution(object):
    def sortList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        # 将两个有序链表合并
        def merge(head1, head2):
            # 排序后的总链表的伪表头
            dummyHead = ListNode()
            # 三个指针，分别指向排序后的总链表、链表1、链表2的相应位置
            temp, temp1, temp2 = dummyHead, head1, head2
            # 开始移动三个指针
            while temp1 and temp2:
                if temp1.val <= temp2.val:
                    temp.next = temp1
                    temp1 = temp1.next
                else:
                    temp.next = temp2
                    temp2 = temp2.next
                temp = temp.next
            # 此时temp1或temp2移到了终点，则将没走到中点的那一个链表剩余的部分直接接到总链表后边
            if temp1:
                temp.next = temp1
            elif temp2:
                temp.next = temp2
            return dummyHead.next


        # 主函数
        def sortListMain(head, tail):
            # 首先通过快慢指针法找到中点
            if not head:
                return head
            # 若该段head和tail直接相连，即为递归的终止条件，应返回
            if head.next == tail:
```

```
                head.next = None # 该句作用是在递归拆分链表时断开左半部分的尾部，确保每个子链表独立，防止合并时出现错误连接

                return head
            slow = fast = head
            while fast != tail:
                slow = slow.next
                fast = fast.next
                if fast != tail:
                    fast = fast.next
            mid = slow
            # 然后递归地分别将两个子链表进行排序
            left_list = sortListMain(head, mid)
            right_list = sortListMain(mid, tail)
            # 最后将两个链表合并
            return merge(left_list, right_list)

        return sortListMain(head, None)
```

- Eg5. 回文链表

  判断一个链表是否是回文的，如`"1,2,2,1"`或`"1,3,2,3,1"`

  解：思路为，首先通过快慢指针法找到链表中点位置，然后将链表后半段进行翻转，最后再依次比较前半段和翻转后的后半段的各个元素是否相等

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        if not head: return False
        if not head.next: return True

        # 首先通过快慢指针确定链表“中点”
        # - 如果链表长度为奇数，则slow最终到达的是mid+1的位置（后半段的起点）
        # - 如果链表长度为偶数，则slow最终到达的是len/2+1的位置（后半段的起点）
        slow, fast = head, head
        while fast:
            slow = slow.next
            fast = fast.next
            if fast:
                fast = fast.next

        # 然后将后半段链表进行翻转
        mid = slow
        pre, cur = None, mid
        while cur:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
        rev_head = pre
```

```
        # 最后依次比较前半段和翻转后的后半段链表的各个节点值是否相等
        cur1, cur2 = head, rev_head
        while cur2:
            if cur1.val != cur2.val:
                return False
            cur1 = cur1.next
            cur2 = cur2.next
        return True
```

- Eg6. 重排链表

  给定一个单链表 `L` 的头节点 `head` ，单链表 `L` 表示为：

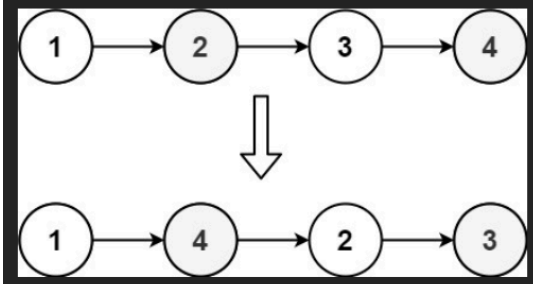
  ```
  L0 → L1 → … → Ln - 1 → Ln
  ```

  请将其重新排列后变为：

  ```
  L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
  ```

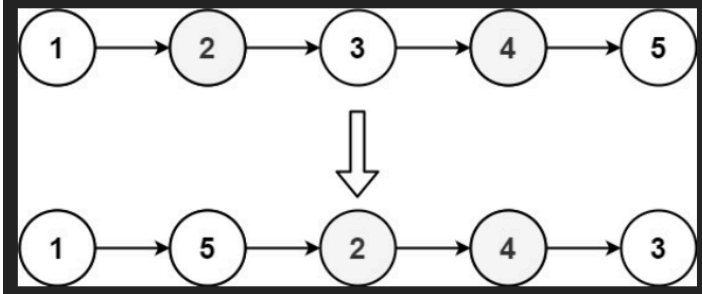
  不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

  示例 1：

  > **输入：** head = [1,2,3,4]
  > **输出：** [1,4,2,3]

  示例 2：

  > **输入：** head = [1,2,3,4,5]
  > **输出：** [1,5,2,4,3]

  解：

该问题可以被拆解成3步：1. 通过快慢指针法找到链表中点；2. 断开前后两段链表的连接后，将后半段链表进行翻转；3. 将前半段链表和翻转后的后半段链表进行交错地合并

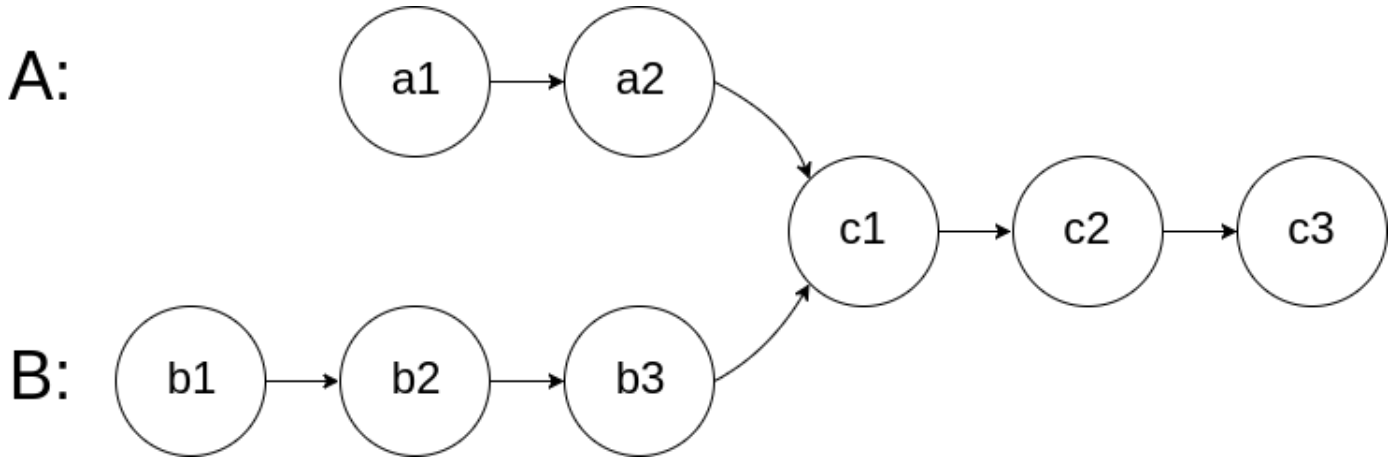
```
    def reorderList(self, head: Optional[ListNode]) -> None:
        """
        Do not return anything, modify head in-place instead.
        """
        # 1. 使用快慢指针法找到链表中点
        slow, fast = head, head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        middle = slow.next
        slow.next = None  # 断开前半段最后一个节点和后半段第一个节点（middle）的连接

        # 2. 翻转后半段链表
        pre, cur = None, middle
        while cur:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
        head_2 = pre   # 翻转后的第二段链表的头节点

        # 3. 交错地合并两段链表
        cur1, cur2 = head, head_2
        while cur1 and cur2:
            tmp1 = cur1.next
            tmp2 = cur2.next
            cur1.next = cur2
            cur2.next = tmp1
            cur1 = tmp1
            cur2 = tmp2

        return head
```

题型：链表交点问题

Eg1：相交链表

给你两个单链表的头节点 `headA` 和 `headB` ，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 `None` 。题目数据 **保证** 整个链式结构中不存在环。

图示两个链表在节点 `c1` 开始相交：



解：使用双指针`pA,pB`，分别从两个链表头部开始同步往后移动。当`pA`为None（到达链表A结尾）时则将其移动到链表B的开头`headB`，当`pB`为None（到达链表B末尾）时则将其移动到链表A的开头`headA`。当移动到`pA == pB`时则可以返回（如果有交叉点则是交叉点，如果没交叉点则是None）

解释：对于有相交的情况，假设链表A的长度为a+c，链表B的长度为b+c，其中c为二者共享的部分。若a==b，则显然`pA,pB`会同时到达交叉点，并返回交叉点。若a不等于b，则二者不会同时到达交叉点，当`pA`移动到链表A末尾时总共移动距离为a+c，然后到达`headB`并移动b后到达交叉点，此时总移动距离为a+c+b；同理`pB`移动b+c+a后也再次到达交叉点，由于此时`pA`和`pB`移动距离相同（a+c+b=b+c+a），所以它们会在交叉点碰面。

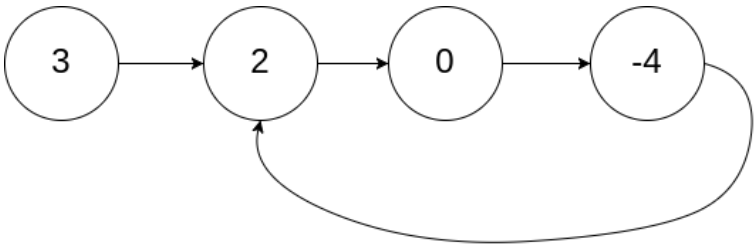
对于无相交的情况，假设两个链表长度为a和b，若a不等于b，则对于`pA`来说，其移动a后到达A末尾，移动到`headB`后再移动b到达B末尾，总共移动距离为a+b；同理`pB`移动b+a后到达A的末尾，此时二者移动总距离相等且都到达None，因此返回也是None。

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> Optional[ListNode]:
        pA, pB = headA, headB
        # if one of them is None then there's no intersection
        if not pA or not pB: return None

        while pA != pB:
            if pA:
                pA = pA.next
            else:   # pA == None
                pA = headB

            if pB:
                pB = pB.next
            else:   # pB == None
                pB = headA
        # when pA == pB, break the loop and return their intersection
        return pA
```

Eg2：环形链表2

给定一个链表的头节点 `head` ，返回链表开始入环的第一个节点。 如果链表无环，则返回 `None`

例如：下图情况即应该返回数值为2的节点



解：和“相交链表”类似，这种寻找距离尾部第k个节点、寻找相交节点、寻找环入口节点等链表题都可使用双指针来解决。

延续普通环形链表中的快慢指针思路，研究fast和slow指针第一次相遇时二者走过的距离：设从head开始链表进入环之前的长度为a，环的长度为b，设fast和slow某一时刻分别已走过f和s个节点，则fast和slow第一次相遇时，fast比slow多走了b个节点（也即一个环的长度）：`f=s+b`。而同时，fast走过的距离是slow的2倍，因此还有：`f=2s`。因此可以解出，第一次相遇时：`s=b, f=2b`。

我们想找到的环入口是从head往后走a或a+b个节点后的节点，而此时slow已走过了b个节点，再走a个节点就能回到环入口处。假设fast和slow相遇时，另一个slow_2节点也从head开始往后走，和slow保持同步，二者都是每次往后走一个节点，则slow_2走a个节点的时候也恰好到达环入口处，此时其恰好和刚走了a+b的slow指针在环入口相遇，因此slow和slow_2相遇时，相遇点就是环入口

```
class Solution:
    def detectCycle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if not head or not head.next: return None
        fast, slow = head, head
        while fast:
            fast = fast.next
            if not fast or not fast.next:    # 无环
                return None
            fast = fast.next
            slow = slow.next
            if fast == slow:    # 第一次相遇，此时slow走了nb
                break
        slow_2 = head
        while slow_2 != slow:
            slow = slow.next
            slow_2 = slow_2.next
        return slow
```

二叉树

二叉树定义：

```
class TreeNode:
    def __init__(self, x):
        self.val = x      # 节点值
        self.left = None  # 左子节点
        self.right = None # 右子节点
```

二叉树的搜索问题见“搜索算法”部分（层序遍历）

二叉树遍历基本方法（前序/中序/后序）

二叉树的遍历方法：

> 前序、中序、后续三种遍历方法都是递归地进行遍历，遍历顺序分别为：根-左-右、左-根-右，左-右-根。代码中唯一区别就在于将此时遍历到的节点加入result列表的时机。

- 前序遍历：

```
# 定义一个树的节点
class TreeNode:
    def __init__(self, x):
        self.val = x  # 值
        self.left = None  # 左节点
        self.right = None  # 右节点


# 前序遍历，result是传入的list型参数，为了递归得到一个遍历List结果
def preorder(root, result):
    result.append(root.val)  # 先添加根节点值到result
    if root.left:
        preorder(root.left, result)  # 递归寻找左子树
    if root.right:
        preorder(root.right, result)  # 递归寻找右子树
    return result
```

前序遍历非递归迭代法（DFS）：

```
    def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root: return []
        result = []
        stack = []
        cur = root
        while stack or cur:
            while cur:
                result.append(cur.val)  # 遍历到当前点，root
                stack.append(cur)   #  将当前点入栈，目的不是用于后续遍历（上一步已经遍历了，而是用于后续索引其右子树）
                cur = cur.left  # 进一步往左下移
            tmp = stack.pop()   # 弹出当前最左下的元素，移动到其右子树
            cur = tmp.right
        return result
```

设置一个栈，以及一个cur指针。循环中，每次先从当前cur开始不停往左下走，且每走一步都访问一下当前节点，并把当前节点入栈（因为前序遍历会先访问root。入栈并不是为了以后访问，而是问了以后访问其右子树）。当cur移到最左下时，弹出当前最左下的元素，并将cur设为其右子树，继续重复。

- 中序遍历：

```
def infix_order(root, result):
    if root.left:
        infix_order(root.left, result)
    result.append(root.val) # 在中间将根节点值添加到result
    if root.right:
        infix_order(root.right, result)
    return result
```

非递归迭代法（DFS）：

```
    def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root: return []

        result = []
        stack = []
        cur = root
        while stack or cur:
            while cur:
                stack.append(cur)    #  一直向左下方移动，但因为遍历方向为左-中-右，所以移动过程中只是入栈而不遍历
                cur = cur.left
            tmp = stack.pop()    #  等到遍历到底的时候，开始出栈访问
            result.append(tmp.val)
            cur = tmp.right # 访问完一个点后移到其右子树
        return result
```

和前序遍历有点类似，也维护一个栈和一个cur节点，每轮循环中将当前节点cur不停往左下移，但中序遍历中每次只入栈而不访问，因为cur相当于“root”的位置，它不应最先被访问，因此目前只是记录。等到最左下时，出栈当前最左下的元素并访问它，然后将cur移到其右子树。

- 后序遍历：

```
def epilogue(root, result):
    if root.left:
        epilogue(root.left, result)
    if root.right:
        epilogue(root.right, result)
    result.append(root.val) # 在最后将根节点值添加到result
    return result
```

非递归迭代法（DFS）

由于后序遍历的顺序是左-右-中，因此按照上述中序遍历的代码风格，对于当前的栈顶`stack[-1]`（当前节点），其左子树肯定已经被访问完毕了，但如果其右子树还没有被访问的话，那么此时就应该先把它的右子树节点全部入栈并进行访问，然后再翻回头来访问当前节点，而不应该直接弹出栈顶然后访问当前节点。

因此，需要额外设置一个`last_visited`来标记上一个被访问到的节点，从而用于判断当前节点的右子树是否已经被访问完毕了。面对当前栈顶`stack[-1]`，首先看一下它是否存在右子树，且上一个被访问到的节点`last_visited`是不是当前栈顶的右子节点：

- 如果存在右子树且右子节点并不是上一个被访问的节点，那就说明其右子树还没被访问，因此本轮迭代先不将当前节点出栈并访问，而是将`cur`切换到其右子节点上，这样以来其右子树的所有节点会被依次入栈和访问，由于这些右子树节点在栈中的位置都会比当前节点更靠栈顶，因此它们也会比当前节点更早被访问到。等它们全都被访问完后，当前节点才会再次变成栈顶。
- 如果不存在右子树那直接弹出当前节点并将其访问和记录为`last_visited`即可
- 如果存在右子树，且右子节点恰好就是`last_visited`，那么说明当前节点的右子树已经被访问完毕了，其右子节点是右子树中最后一个被访问到的，它被访问完后当前节点再次浮上来变成了栈顶。此时就可以将栈顶弹出并访问，并将其标记为`last_visited`了

```
    def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        stack = []
        result = []
        cur = root
        last_visited = None

        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.left

            # 当前节点的左子树此时已经访问完毕了
            # 对于当前节点自己而言，先不将其出栈，先看看它的右子树是否已经被访问过了
            tmp = stack[-1]
            # 如果当前节点存在右子树，且其右子节点不是上一个被访问的节点，那么说明其右子树还没有被访问
            # 那么就先不将当前节点出栈并访问，而是将cur切换到其右子节点上，并将其右子树入栈
            # 这样就可以确保其右子树节点都比其更接近栈顶，因此能比当前节点更先被访问到
            if tmp.right and tmp.right != last_visited:
                cur = tmp.right
            # 如果当前节点不存在右子树，那就直接出栈并访问当前节点即可
            # 或者如果当前节点存在右子树，但其右子节点就是上一个被访问到的节点，那说明当前节点的右子树刚刚被访问完毕，现在轮到当前节点被访问了
            # 此时将当前节点出栈，并访问当前节点，且将其记录为上一个被访问到的节点
            else:
                tmp = stack.pop()
                result.append(tmp.val)
                last_visited = tmp

        return result
```

二叉树遍历的应用：

Eg1. 验证二叉搜索树

给你一个二叉树的根节点 `root` ，判断其是否是一个有效的二叉搜索树。

有效 二叉搜索树定义如下：

- 节点的左子树只包含 **小于** 当前节点的数。
- 节点的右子树只包含 **大于** 当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

解：

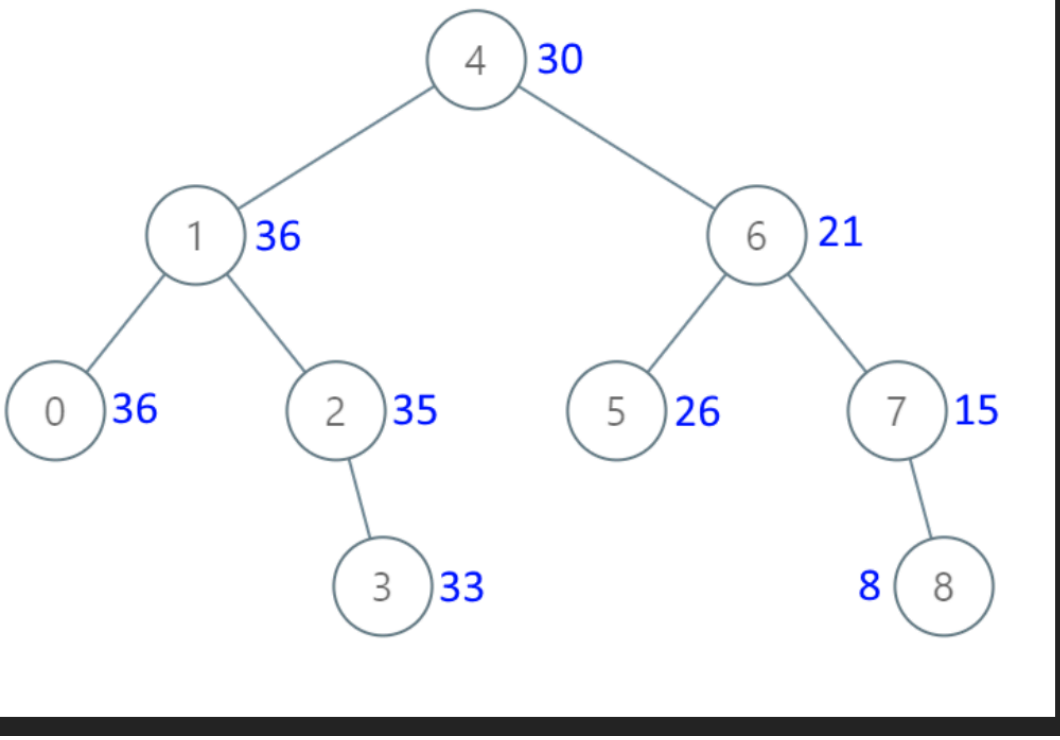
二叉搜索树的定义等价于：该二叉树的中序遍历结果一定是单调递增的。因此，只需将其进行中序遍历，每次比较当前遍历到的节点值是否大于上一个遍历到的节点值即可

```
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        if not root.left and not root.right: return True
        last_node = None
        stack = []
        cur = root
        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.left
            tmp = stack.pop()
            if last_node:
                if last_node.val < tmp.val:
                    last_node = tmp
                else:
                    return False
            else:
                last_node = tmp
            cur = tmp.right
        return True
```

Eg 2. 把二叉搜索树转换为累加树

给出二叉 **搜索** 树的根节点，该树的节点值各不相同，请你将其转换为累加树（Greater Sum Tree），使每个节点 `node` 的新值等于原树中大于或等于 `node.val` 的值之和。

示例 1：



解：

本题只需逆向中序遍历（右-中-左）这个二叉搜索树，使得遍历到的节点值从大到小，并在遍历到每个节点时，将其本身值加到全局累加器上，然后再将此时的全局累加器的值赋予该节点即可

```
class Solution:
    def convertBST(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root: return root

        stack = []
        cur = root
        sum_ = 0

        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.right
            tmp = stack.pop()
            sum_ += tmp.val
            tmp.val = sum_
            cur = tmp.left
        return root
```

Eg3. 二叉树展开为链表

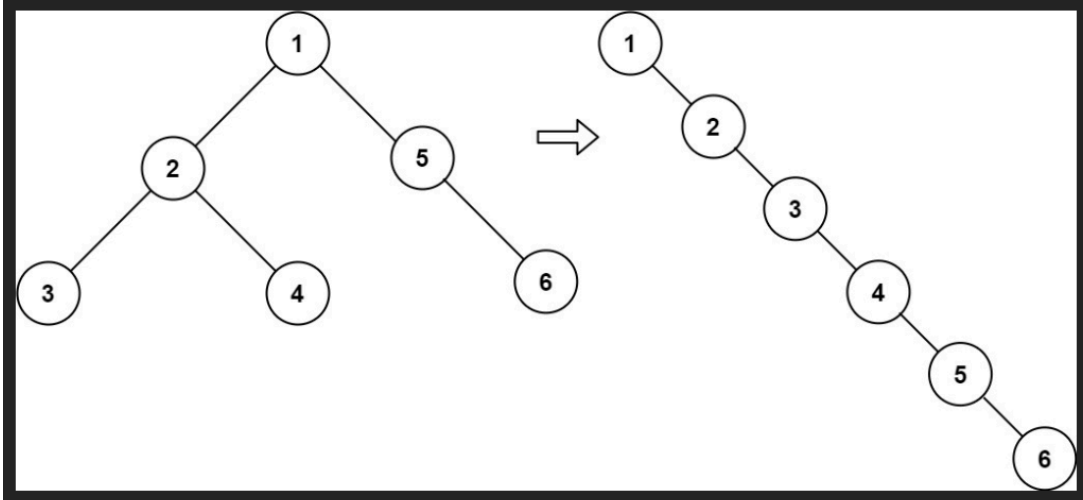
给你二叉树的根结点 `root` ，请你将它原地展开为一个单链表：

- 展开后的单链表应该同样使用 `TreeNode` ，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `None` 。
- 展开后的单链表应该与二叉树 先序遍历**先序遍历**顺序相同。

函数不需要返回任何值，展开后的链表仍连接在原二叉树的root节点上。

使用$O(1)$空间复杂度完成。

示例 1：




```
输入：root = [1,2,5,3,4,null,6]
输出：[1,null,2,null,3,null,4,null,5,null,6]
```

解：

由于要求O(1)空间复杂度，因此不能先前序遍历将结果储存在链表中然后再重构链表。因此，这里借鉴后续遍历中保存前一个节点的思路，使用一种特殊算法（Morris遍历）来原地修改树的结构。

简单来说，由于前序遍历顺序为根-左-右，因此在最终构造出的链表中，根-左子树-右子树也是依次往右下线性排列的，因此一个基本的思路是：将每个节点的左子树“插入到”当前节点与其右子节点之间，这样的话在链表中不断往右下走，访问到的顺序就是根-左子树-右子树了。

具体而言，每遍历到一个节点时，如果它存在左子树的话，首先找到该左子树中的最靠右边的节点——在前序遍历中，它是左子树里最后一个被遍历到的节点，遍历完它后的下一个被遍历节点就是右子树的第一个节点（右子树根节点）了。因此，找到左子树最右边的节点后，将它和右子树根节点进行连接，也即将它的右子节点设为右子树根节点，这样左-右子树就串起来了。最后，再把左子树整个移动到当前节点的右子树位置，并把左子节点置为 `None`，此时根-左-右就串成了。

当前节点处理完毕后，继续往下处理它的右子节点（`cur = cur.right`）

```python
class Solution:
    def flatten(self, root: Optional[TreeNode]) -> None:
        """
        Do not return anything, modify root in-place instead.
        """
        if not root: return root
        cur = root
```

```
        while cur:
            # 如果当前节点存在左子树，则：
            # 1. 找到其左子树最右边的节点
            # 2. 将其左子树最右边节点的右子节点设为当前节点的右子节点，也即将右子树整个连在左子树后边
            # 3. 将当前节点的左子节点（整个左子树）移动到其右子节点
            if cur.left:
                cur_left_rightmost = cur.left
                # 首先找到左子树最右边的节点
                while cur_left_rightmost.right:
                    cur_left_rightmost = cur_left_rightmost.right

                # 此时，cur_left_rightmost为左子树最右边的节点，然后将当前节点的右子节点设为其右子节点，使得左子树末尾和右子树开头连接上
                cur_left_rightmost.right = cur.right

                # 最后，将当前节点整个左子树都移动到其右子树位置，并将左子树置为None
                cur.right = cur.left
                cur.left = None

            # 然后继续向右下移动当前节点指针
            cur = cur.right
```

Eg4. 从前序与中序遍历序列构造二叉树

给定两个整数数组 `preorder` 和 `inorder` ，其中 `preorder` 是二叉树的**先序遍历**列表， `inorder` 是同一棵树的**中序遍历**列表，请构造二叉树并返回其根节点。（保证树中所有节点值都各不一样）

解：

设置一个构造树的递归函数，给定这个树（子树）的前序序列（子段）和中序序列（子段），构造它并返回它的根节点。

核心突破点在于根节点。给定前序序列，则当前子树的根节点一定是它的第0个元素，这就得到了当前构造树的根节点值。由于保证树中节点值各不一样，因此可以根据根节点值进一步在中序序列中识别出它所在位置，则中序序列中它左边的元素都属于左子树，右边的元素都属于右子树，这样就可以分别得到左子树和右子树在中序序列中对应的子段起始点和终点。

进一步，中序序列中由根节点分隔后，还可以得到左子树和右子树的节点数量，这样就可以返回来从前序序列中识别出左子树和右子树对应的子段（因为前序序列的顺序是根-左-右，之前不知道左右子树的分界点在哪里，而现在知道了左右子树的大小就可以知道分界点了）。

至此，当前构造树的根节点的左、右子树在前序、中序序列中对应的子段始末位置都已确定，则可以进一步递归地调用构造函数来构造左右子树。最终返回当前构造树的根节点。

```
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        # 为了方便从中序遍历的序列中查找root所在的位置，构建中序遍历序列的val:idx哈希表，以便每次能通过root.val值快速找到其在中序序列中的位置
        # 前提：节点和值一一对应，不存在值相等的节点
        inorder_map = {val: idx for idx, val in enumerate(inorder)}
```

```
        # 定义构建树的递归函数
        # 其负责构建pre_start ~ pre_end区间内对应的子树（同时也是in_start~in_end区间内的子树），并返回该子树的根节点
        def buildTree_(pre_start, pre_end, in_start, in_end):
            # 跳出条件：区间左端点位置越过了右端点
            if pre_start > pre_end:
                return None

            # 首先，构建该子树的根节点，其一定是当前处理的preorder序列中的第一个元素
            root_val = preorder[pre_start]
            root = TreeNode(val=root_val)

            # 然后，找一下当前子树的根节点在对应的inorder序列中的位置
            root_idx_inorder = inorder_map[root_val]

            # 找到根节点在inorder序列中的位置后，则inorder序列中它左边的就是它的左子树，右边的就是它的右子树。也即，可以在inorder序列中通过root节点来分隔开左子树和右子树，找到它们对应的inorder区间段
            left_start_inorder = in_start
            left_end_inorder = root_idx_inorder - 1
            right_start_inorder = root_idx_inorder + 1
            right_end_inorder = in_end
            # 进一步可以求出左子树和右子树分别有多少个节点，然后根据节点数量来在preorder序列中识别出对应的左子树和右子树区间
            left_size = root_idx_inorder - in_start
            right_size = in_end - root_idx_inorder

            # 知道左子树和右子树节点数量后，由于preorder序列中的顺序是根-左-右，因此即可进一步识别出左右子树在preorder序列中对应的区间段
            left_start_preorder = pre_start + 1
            left_end_preorder = pre_start + left_size
            right_start_preorder = pre_start + left_size + 1
            right_end_preorder = pre_end


            # 此时已经分别识别出了左右子树在preorder、inorder中对应的区间段，因此可以递归调用构造函数来分别构造左右子树
            root.left = buildTree_(left_start_preorder, left_end_preorder, left_start_inorder, left_end_inorder)
            root.right = buildTree_(right_start_preorder, right_end_preorder, right_start_inorder, right_end_inorder)

            # 构造完毕后返回当前子树根节点
            return root

        return buildTree_(0, len(preorder)-1, 0, len(inorder)-1)
```

二叉树层序遍历

二叉树层序遍历基于BFS实现

基础层序遍历：

- eg1：从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。例如给定二叉树：`[3,9,20,null,null,15,7]`

  ```
    3
   / \
  9  20
    /  \
   15   7
  ```

  返回：[3,9,20,15,7]

  解答：

  使用队列来确保先进先出（二叉树不需要标记访问，因为每个节点只可能有一个父亲节点）

  ```python
  # Definition for a binary tree node.
  # class TreeNode:
  #     def __init__(self, x):
  #         self.val = x
  #         self.left = None
  #         self.right = None

  class Solution:
      def levelOrder(self, root: TreeNode) -> List[int]:
          if not root: return []
          res, queue = [], collections.deque()
          # 将根节点入队
          queue.append(root)
          # 开始循环，直到队列为空，表示已经遍历了整个二叉树
          while queue:
              # 将此时的队首出列
              node = queue.popleft()
              # 收集出列的队首的值，并记录在结果中
              res.append(node.val)
              # 将左节点入队（如存在）
              if node.left:
                  queue.append(node.left)
              # 将右节点入队（如存在）
              if node.right:
                  queue.append(node.right)
          return res
  ```

- eg2：从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

  例如对于上问示例，应返回：

```
[
  [3],
  [9,20],
  [15,7]
]
```

解答：

要将每一层单独存储，只需在上题基础上利用一个特点：队列长度。让每轮`while queue`循环均只保存一层的数据，则只需在该轮内部再添加一个存储该层的临时列表`result_layer`，且在内部再添加一个对此时队列长度的循环即可，然后将上题逻辑搬到这个内层循环中，且加入结果加入的是该层临时列表`result_layer`：

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root: return []
        result = []
        queue = collections.deque()
        queue.append(root)
        while queue:
            result_layer = []
            for _ in range(len(queue)):
                node = queue.popleft()
                result_layer.append(node.val)
                if node.left: queue.append(node.left)
                if node.right: queue.append(node.right)
            result.append(result_layer)

        return result
```

- eg3：按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

  例如对于上问示例，应返回：

```
[
  [3],
  [20,9],
  [15,7]
]
```

解答：

在上题基础上（每轮`while queue`循环均只保存一层的数据），只需设置一个标志位flag使其每轮`while queue`都翻转一次来标记本层是应该正序存放还是倒序存放，然后将保存该层的临时列表`result_layer`改成双端队列，当`flag==False`时使用`appendleft()`函数来将出队（`queue`）元素从左插入该临时列表。注意将左右节点加入`queue`时顺序不变，还是先左后右。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root: return []
        result = []
        queue = collections.deque()
        queue.append(root)
        flag = True
        while queue:
            result_layer = collections.deque()
            for _ in range(len(queue)):
                node = queue.popleft()
                if flag:
                    result_layer.append(node.val)
                else:
                    result_layer.appendleft(node.val)

                if node.left: queue.append(node.left)
                if node.right: queue.append(node.right)

            flag = not flag
            result.append(list(result_layer))

        return result
```

- Eg4: 二叉树的最大深度

  给定一个二叉树 `root` ，返回其最大深度。

  二叉树的 **最大深度** 是指从根节点到最远叶子节点的最长路径上的节点数。

  解：

  使用层序遍历来统计树的层数即可

```
class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        if not root: return 0

        # 层序遍历
        from collections import deque
        queue = deque()
```

```
        queue.append(root)
        cnt = 0
        while queue:
            result_layer = []
            for i in range(len(queue)):
                node = queue.popleft()
                result_layer.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)
            cnt += 1
        return cnt
```

> 还可以直接使用递归来计算：以`root`为根的树的高度等于其左子树和右子树中高度较大的那个再+1（加上根节点本身）
>
> ```
> def height(self, root):
>     if not root:
>         return 0
>     # 以root为根的树的高度为其左子树高度和右子树高度中较大的那个再+1（根节点本身）
>     return max(self.height(root.left), self.height(root.right)) + 1
> ```

题型：通过节点index来感知二叉树结构

> 该类问题通常需要考虑非满二叉树的null部分，而普通的层序遍历只能搜索到真实的节点，无法感知二叉树的结构信息，因此无法求出二叉树宽度、重构二叉树结构等。此时可以在队列状态中多添加一项节点编号（index）来记录节点在二叉树结构中的相对位置信息，重要性质为：编号为`i`的节点的左右子节点的index分别是`2*i`和`2*i+1`。

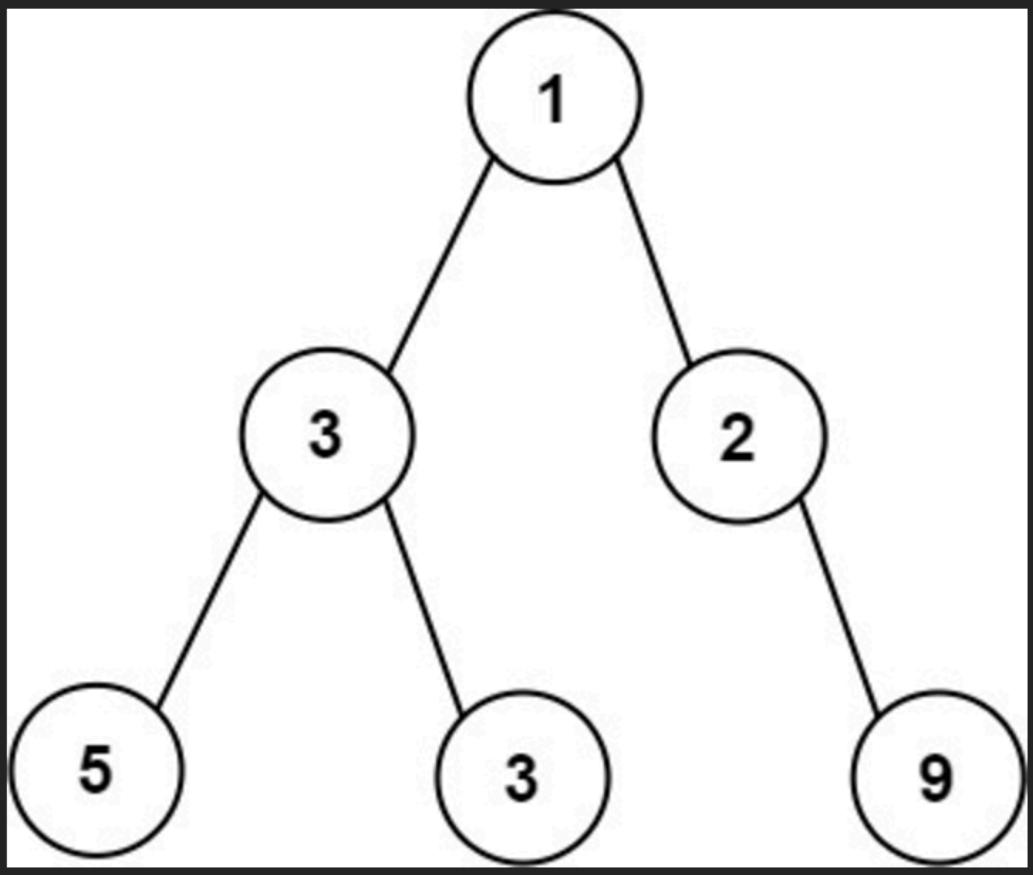
Eg1 二叉树的最大宽度

给你一棵二叉树的根节点 `root` ，返回树的 **最大宽度** 。

树的 **最大宽度** 是所有层中最大的 **宽度** 。

每一层的 **宽度** 被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 `null` 节点，这些 `null` 节点也计入长度。

示例 1：




```
输入：root = [1,3,2,5,3,null,9]
输出：4
解释：最大宽度出现在树的第 3 层，宽度为 4 (5,3,null,9) 。
```

解：

显然也应该以层序遍历为基础。但是不能直接套用层序遍历逐层的结果，因为这样会跳过中间空缺的节点使之无法追踪。因此，可以通过在节点状态中增加一项“index”来表征每个节点在二叉树中的绝对位置。

具体而言，若某个节点的index为`i`（设根节点的index为`1`），则其左子节点的index为`2*i`，右节点的index为`2*i+1`。这样就可以表征每个节点的绝对位置，每一层两端的节点index之差即为该层的长度。

```python
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        if not root.left and not root.right:
            return 1

        result = []

        import collections
        queue = collections.deque()
        queue.append((root, 1)) # 给每个节点编号，节点i的左子节点编号为2*i，右节点编号为2*i+1
```

```
        max_res_layer = 1

        while queue:
            result_layer = []
            for _ in range(len(queue)):
                node, index = queue.popleft()
                result_layer.append(index)
                if node.left:
                    queue.append((node.left, index * 2))
                if node.right:
                    queue.append((node.right, index * 2 + 1))
            # 每层记录该层所有节点的编号，该层最大编号减去最小编号就是该层宽度
            max_res_layer = max(max_res_layer, result_layer[-1] - result_layer[0] + 1)
        return max_res_layer
```

Eg2 二叉树的序列化与反序列化

序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。也即，给定一个二叉树的`root`节点，`serialize()`函数将该二叉树转换为一个字符串（也即“编码”），然后`deserialize()`函数需要基于这个字符串完全还原二叉树，并返回还原后二叉树的`root`节点。

解：

由于普通的前/中/后/层序遍历方法只能遍历到各个节点本身，对于非满二叉树无法完整记录其结构信息（因为存在null节点）。因此，类似上题在层序遍历的队列节点状态中添加一项index，即可记录各个节点在二叉树中的相对位置信息，这样就可以在反序列化时将二叉树进行还原。

具体而言，编码函数中，设置一个`index_map`字典，用于记录各个index值和节点val值的对应关系，然后层序遍历原二叉树并将每个节点的index和val信息记录进`index_map`中，最后再将`index_map`字典编码为一个字符串作为编码结果。解码函数中，首先根据字符串还原`index_map`字典，然后构建出`root`节点（也即index=1的节点），然后再从这个重建的`root`节点开始层序遍历：对于遍历到的当前节点，设其index为`i`，则通过查看`2 * i`和`2 * i + 1`是否存在于`index_map`中，来确认其左右子节点是否存在，如果存在的话则利用`index_map[2*i]`和`index_map[2*i+1]`的信息获取二者的val，并构建出左右子节点，将它们连接到当前节点上，并加入队列以便进一步遍历。最终即可完成二叉树的重建。

```
class Codec:

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        if not root:
            return ""
```

```
        # index_map用于保存节点index与节点value的对应关系，用于重建二叉树
        # key为节点index，value为节点value
        index_map = {}

        # 层序遍历，将二叉树的节点与结构信息记入index_map
        from collections import deque
        queue = deque()
        queue.append((root, 1)) # root节点的index=1
        while queue:
            node, index = queue.popleft()
            index_map[index] = node.val     # 将当前节点的index和value信息记入index_map
            if node.left:
                queue.append((node.left, 2 * index))
            if node.right:
                queue.append((node.right, 2 * index + 1))

        # 将index_map编码为一个字符串："index_1:val_1,index_2:val_2,..."
        data = ",".join(f"{index}:{val}" for index, val in index_map.items())
        return data


    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        if not data:
            return None

        # 从字符串还原回index_map
        index_map = {}
        for item in data.split(','):
            index, val = item.split(':')
            index_map[int(index)] = int(val)

        # 通过层序遍历来还原二叉树（从root开始，以层序遍历的顺序来依次生长出各个子节点）
        from collections import deque
        queue = deque()
        root = TreeNode(index_map[1])   # 编号为1的节点是root节点，首先将它还原
        queue.append((root, 1))

        while queue:
            node, index = queue.popleft()
            # 推导出当前节点的左子节点和右子节点的index
            index_left = 2 * index
            index_right = 2 * index + 1
            # 通过查找两个子节点的index是否在index_map中，来判断原二叉树中当前节点是否存在左右子节点，
如果存在的话则将它们构建出来并连接到当前节点上，然后入队以备进一步访问
            if index_left in index_map.keys():
                node.left = TreeNode(index_map[index_left]) # 将其连接到当前节点上
                queue.append((node.left, index_left))   # 将其入队
```

```
            if index_right in index_map.keys():
                node.right = TreeNode(index_map[index_right]) # 将其连接到当前节点上
                queue.append((node.right, index_right))    # 将其入队

        return root
```

题型：通过层序遍历构造新二叉树

Eg1. 合并二叉树

给你两棵二叉树： `root1` 和 `root2` 。

想象一下，当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠（而另一些不会）。你需要将这两棵树合并成一棵新二叉树。合并的规则是：如果两个节点重叠，那么将这两个节点的值相加作为合并后节点的新值；否则，**不为** None 的节点将直接作为新二叉树的节点。

返回合并后的二叉树。

注意: 合并过程必须从两个树的根节点开始。

解：

使用层序遍历来同步地遍历三棵树，也即每次加入queue的都是3个当前节点，这三个当前节点在二叉树中的相对位置是一样的。

每遍历到一个节点位置时，如果两个给定树的当前节点都有左子节点，那么合并树的当前节点也应该有左子节点，且节点值为二者相加（如果有一方没有的话就当成0），然后即可构造一个合并节点并将其接到合并树当前节点的左子节点上。然后进一步把三个左子节点（在二叉树中的相对位置同样是相同的）入队，如果某个给定树的当前节点没有左子节点（也即其给合并节点的贡献值为0），为了保持三棵树遍历的同步性，它也需要入队一个值为0的“占位符”节点。对于右子节点的处理是完全一样的。

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def mergeTrees(self, root1: Optional[TreeNode], root2: Optional[TreeNode]) ->
Optional[TreeNode]:
        if not root1: return root2
        if not root2: return root1

        root = TreeNode(val=root1.val+root2.val)    # 合并后的新树的root

        from collections import deque
        queue = deque()
        queue.append((root, root1, root2))    # 用一个queue，同步遍历三个树

        while queue:
```

```
            cur, cur1, cur2 = queue.popleft()

            # 处理左子节点:
            # 如果cur1或cur2存在左子节点，则cur也应该具有这个左子节点，这种情况是需要处理的
            # 若只有二者之一具有左子节点，为了保持三棵树遍历位置的同步性，对于没有左子节点的那一方，也需要入队一个值为0的“占位符”左子节点
            if cur1.left or cur2.left:
                # 取二者的值相加（如果某一边没有左子节点就当成0），构造一个合并后的新节点，并连接到合并树上
                val1 = cur1.left.val if cur1.left else 0
                val2 = cur2.left.val if cur2.left else 0
                left_merged_node = TreeNode(val=val1+val2)
                cur.left = left_merged_node

                # 然后，将三棵树的左子节点入队。若其中某个树没有左子节点的话，就入队一个val=0的占位符
                queue.append(
                    (
                        cur.left,
                        cur1.left if cur1.left else TreeNode(val=0),
                        cur2.left if cur2.left else TreeNode(val=0),
                    )
                )

            # 同样方法处理右子节点
            if cur1.right or cur2.right:
                val1 = cur1.right.val if cur1.right else 0
                val2 = cur2.right.val if cur2.right else 0
                right_merged_node = TreeNode(val=val1+val2)
                cur.right = right_merged_node

                queue.append(
                    (
                        cur.right,
                        cur1.right if cur1.right else TreeNode(val=0),
                        cur2.right if cur2.right else TreeNode(val=0),
                    )
                )

        return root
```

题型：二叉树的操作与性质判定

- eg1：二叉树翻转

  使用递归即可，对于root来说，分别对其左右子树进行翻转，然后再将左、右子树分别的根节点互换，即可完成root子树的左右翻转。

  ```
  # Definition for a binary tree node.
  ```

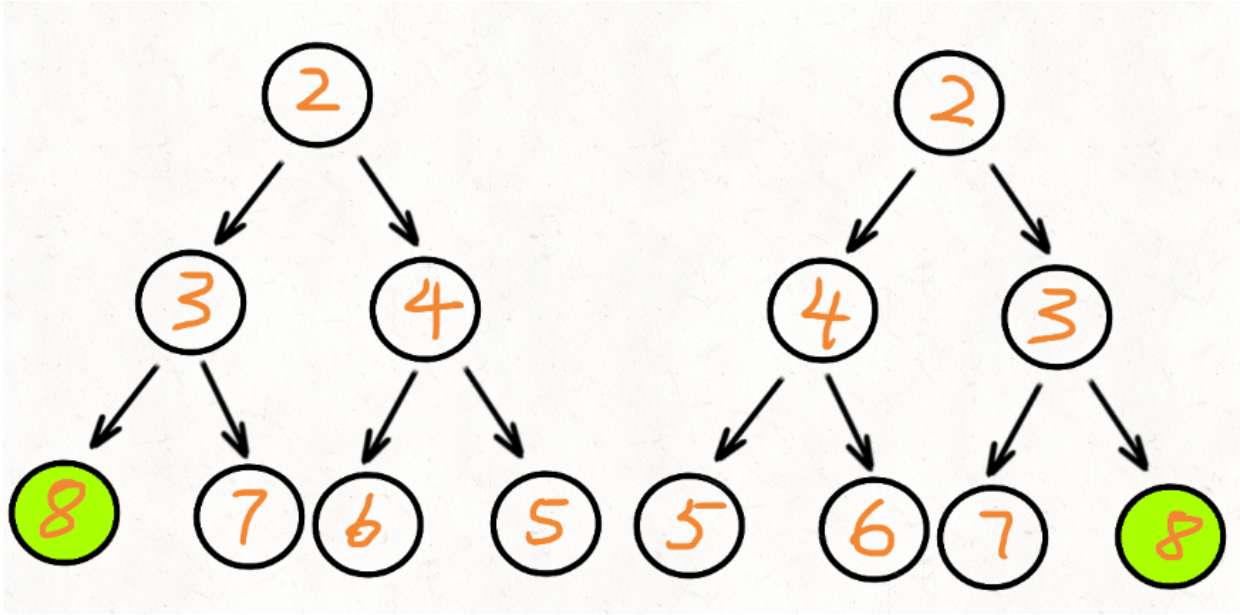
```
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution(object):
    def invertTree(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        if not root:
            return root

        # 递归地翻转左子树
        left_reversed = self.invertTree(root.left)
        # 递归地翻转右子树
        right_reversed = self.invertTree(root.right)
        # 将翻转后的左、右子树的根节点交换，完成root下的左右子树交换
        root.left, root.right = right_reversed, left_reversed
        return root
```

非递归做法：层序遍历，每遍历到一个点就将其左右子树反转：

```
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return root

        # 层序遍历法：每遍历到一个点，就将其左右子树反转
        from collections import deque
        queue = deque()
        queue.append(root)
        while queue:
            node = queue.popleft()
            node.left, node.right = node.right, node.left
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        return root
```

- eg2：对称二叉树

  判断一个二叉树是否为对称的（需要左右两边对应节点的值相等），如：



解：

定义递归调用的`check`函数，它接收`left,right`两个节点作为参数，注意这里除了`root.left, root.right`以外，`left,right`指的都不是一个节点的左右子节点，而是关于二叉树对称轴对称的两个点（例如上边的两个3、两个4）。在函数内部判断`left`和`right`的关系：若二者皆空则返回True；若二者只有一个空则返回False；若二者皆不空但值不等则二者分别对应的子树也不可能对称，返回False；若二者皆不空且值相等，则进一步递归地判断`left.left`与`right.right`、`left.right`与`right.left`所对应的子树是否对称。

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution(object):
    def isSymmetric(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        if not root: return True

        # 递归地检查左边和右边的对应点分别的子树是否对称。注意left和right未必是一个点的左右子节点！它们是整个树关于中轴的对称点，如left.left和right.right
        def check(left, right):
            # 若左右两个节点都为空，则左右两个节点对应的子树对称
            if not left and not right: return True
            # 若左右两个节点有一个是空而一个非空，则左右两个节点对应的子树非对称
            if ((not left) and right) or ((not right) and left): return False
            # 若左右两个节点都非空，但它们本身就不相等，则左右两个节点对应的子树非对称
            if left.val != right.val: return False

            # 若左右两个节点值相等，则它们分别对应的子树有可能是对称的，进一步递归判断即可
            return check(left.left, right.right) and check(left.right, right.left)
```

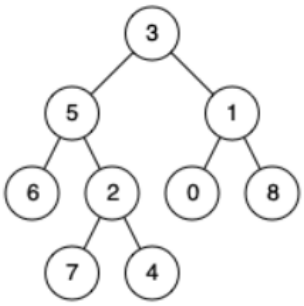
```
        return check(root.left, root.right)
```

- eg3：二叉树的最近公共祖先

  给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

  最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（**一个节点也可以是它自己的祖先**）。”

  示例 1：

  > **输入：** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
  > **输出：** 3
  > **解释：** 节点 `5` 和节点 `1` 的最近公共祖先是节点 `3` 。

  解：

  使用递归。函数本身返回值的含义定义为：以root为根的子树中，p和q的最近公共祖先。如果子树中只有其中一个节点存在，则返回该节点

  - 若root==None，则显然root子树中不存在p,q的公共祖先，返回None
  - 若root==p，则到达一个末端，直接返回p，使之往顶端浮动
  - 若root=q，则到达一个末端，直接返回q，使之往顶端浮动
  - 若root并非p,q，则分别求其左右子树中p,q的最近祖先，使用递归
    - 若左右子树返回值均非None，则说明p,q分别处于左,右子树中，说明root就是分叉的地方，直接返回root作为答案
    - 若只有左子树返回值非None，右子树返回值为None，则说明p,q都聚集于左子树中，因此其最近公共祖先也一定在左子树，故返回左子树的返回值，使之向顶端浮动
    - 若只有右子树返回值非None，同理

```
class Solution(object):
    def lowestCommonAncestor(self, root, p, q):
        """
        :type root: TreeNode
        :type p: TreeNode
        :type q: TreeNode
        :rtype: TreeNode
        """
        if not root or root == p or root == q:
```

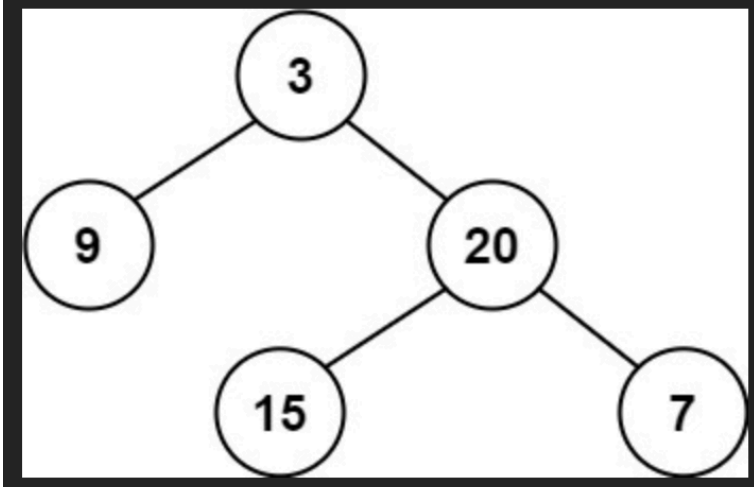
```
            return root

        # 2. 分别递归地在左子树和右子树中查找
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        # 3. 根据递归结果，进行判断
        if left and right:
            # 如果左子树和右子树都找到了节点，说明当前 root 就是最近公共祖先
            return root
        elif left:
            # 如果只有左边找到了，那说明两个节点都在左子树
            return left
        else:
            # 否则就只能在右子树中
            return right
```

- Eg4. 平衡二叉树

  给定一个二叉树，判断它是否是 平衡二叉树。**平衡二叉树** 是指该树所有节点的左右子树的高度相差不超过 1。

  示例：

  示例 1：

  输入： root = [3,9,20,null,null,15,7]
  输出： true

  解：

  对于一个根节点root来说，它对应的树是平衡二叉树的条件是：1、其左右子树高度差距<=1；2、其左子树本身是平衡二叉树；3、其右子树本身是平衡二叉树。

  因此，首先进一步定义一个求树高度的函数`def height()`，其既可以用上文的层序遍历方法求，也可以通过自身递归求：以root为根的树的高度为其左子树高度和右子树高度中较大的那个再+1（根节点本身）。对于主函数`def isBalanced()`来说，根据上一段分析出的三个条件，它也可以写成递归形式：当前节点为根的树是平衡二叉树<=>其左、右子树高度差距<=1（使用`height()`函数求出）且其左右子树分别都是平衡二叉树（调用自身递归求出）

```
```

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def height(self, root):
        if not root:
            return 0
        # 以root为根的树的高度为其左子树高度和右子树高度中较大的那个再+1（根节点本身）
        return max(self.height(root.left), self.height(root.right)) + 1

    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True
        return abs(self.height(root.left) - self.height(root.right)) <= 1 and
self.isBalanced(root.left) and self.isBalanced(root.right)
```

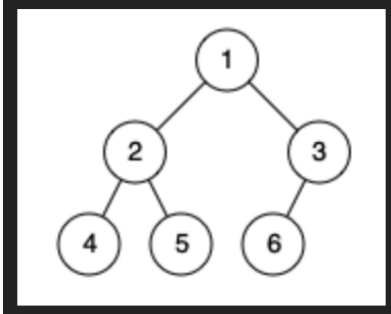
- Eg5. 二叉树的完全性检验

  给你一棵二叉树的根节点 `root` ，请你判断这棵树是否是一棵 **完全二叉树** 。

  在一棵 完全二叉树 中，除了最后一层外，所有层都被完全填满，并且最后一层中的所有节点都尽可能靠左。最后一层（第 `h` 层）中可以包含 `1` 到 `2^h` 个节点。

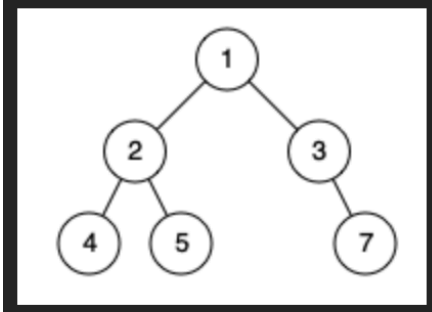
  示例：

示例 1：



> **输入：** root = [1,2,3,4,5,6]
> **输出：** true
> **解释：** 最后一层前的每一层都是满的（即，节点值为 {1} 和 {2,3} 的两层），且最后一层中的所有节点（{4,5,6}）尽可能靠左。

示例 2：



> **输入：** root = [1,2,3,4,5,null,7]
> **输出：** false
> **解释：** 值为 7 的节点不满足条件「节点尽可能靠左」。

解：

层序遍历可以确保每层都是从左往右遍历的，因此可以基于层序遍历完成本问题。当某一层从左到右遍历过程中发现了一个空节点，则如果是完全二叉树的话，本层为最后一层，其之后（本层右侧）应该都是空节点，如果后边还出现了非空节点则说明该树不是完全二叉树。因此，维护一个变量`found_null`表示目前是否发现了空节点，然后开始层序遍历，注意为了追踪空节点的出现，这里每遍历到一个非空节点时不管它是否存在左右节点，都将左右节点入队，也即队列中可能存在空节点。遍历过程中，如果发现了空节点则记录`found_null=True`，然后继续遍历，如果之后又发现了非空节点的话则说明破坏了完全二叉树的结构，直接返回False即可。如果遍历完未发现冲突，则返回True。

```
class Solution:
    def isCompleteTree(self, root: Optional[TreeNode]) -> bool:
        from collections import deque

        if not root:
            return True

        queue = deque()
        queue.append(root)
```

```
        found_null = False  # 当前是否发现了空节点
        while queue:
            node = queue.popleft()

            if not node:        # 如果发现当前节点为空节点，则标记发现了空节点
                found_null = True
            else:   # 如果当前节点不是空节点
                # 如果之前已经发现了空节点，而当前节点非空节点，则说明不是完全二叉树
                if found_null:
                    return False
                # 将当前节点的左右子节点（可能是空节点）加入队列
                queue.append(node.left)
                queue.append(node.right)
        return True
```

题型：二叉树的DFS路径问题

常用于“规定方向从上到下”的路径问题

- Eg4. 路径总和

  给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum` 。判断该树中是否存在 **根节点到叶子节点** 的路径，这条路径上所有节点值相加等于目标和 `targetSum` 。如果存在，返回 `true` ；否则，返回 `false` 。

  叶子节点 是指没有子节点的节点。

  解：

  直接对二叉树使用DFS即可，因为DFS每一条遍历到叶子节点的路径都是从root开始直插往下到底的。放入stack的每个节点都额外保存一个current_sum状态，也即从root到当前节点的路径总和达到了多少。当发现当前节点为叶子节点时则看一下current_sum是否为targetSum，如果发现这样一个叶子节点则返回True，若遍历完了都还没发现这样的叶子节点则返回False

```
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        if not root:
            return False

        stack = [(root, root.val)]

        while stack:
            node, current_sum = stack.pop()

            # 如果是叶子节点且当前和为targetSum，返回True
            if not node.left and not node.right and current_sum == targetSum:
                return True

            if node.left:
                stack.append((node.left, current_sum + node.left.val))
```

```
            if node.right:
                stack.append((node.right, current_sum + node.right.val))

        return False
```

Eg4.2：路径总和II

给你二叉树的根节点 `root` 和一个整数目标和 `targetSum` ，找出所有 **从根节点到叶子节点** 路径总和等于给定目标和的路径。

解：本问题需要将所有和为目标值的路径进行记录，只需在加入stack的节点上再额外保存一个列表，用于储存root到该节点的路径即可。注意列表深浅复制问题防止干扰。

```
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:
        if not root: return []

        stack = [(root, root.val, [root.val])]
        all_paths = []
        while stack:
            node, current_sum, path = stack.pop()

            if not node.left and not node.right and current_sum == targetSum:
                all_paths.append(path)

            if node.left:
                path_ = path[:]
                path_.append(node.left.val)
                stack.append((node.left, current_sum+node.left.val, path_))
            if node.right:
                path_ = path[:]
                path_.append(node.right.val)
                stack.append((node.right, current_sum+node.right.val, path_))
        return all_paths
```

Eg4.3：路径总和III

给定一个二叉树的根节点 `root` ，和一个整数 `targetSum` ，求该二叉树里节点值之和等于 `targetSum` 的 **路径** 的数目。

路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

解：由于路径的要求仍为必须从上往下，所以整体思路上仍为DFS。但由于此时不要求路径非要从root开始，所以保存从root到当前节点的累加和current_sum就没有意义了，只需在stack中保存当前节点以及从root到当前节点的路径。然后遍历到每个节点时（因为不要求路径末端为叶子节点，所以每个节点都有可能作为目标和路径的末端），都逐个验证root到当前节点路径的各个子路径（路径首端从下往上依次遍历，路径末端保持为当前节点），如果找到满足目标和的子路径则count+=1。

```
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        if not root:
```

```
            return 0

        count = 0
        stack = [(root, [root.val])]

        while stack:
            node, path = stack.pop()

            # 检查当前路径的所有可能子路径
            current_sum = 0
            for i in range(len(path)-1, -1, -1):
                current_sum += path[i]
                if current_sum == targetSum:
                    count += 1

            if node.left:
                stack.append((node.left, path + [node.left.val]))
            if node.right:
                stack.append((node.right, path + [node.right.val]))

        return count
```

题型：二叉树+动态规划

> 常见于在二叉树中求最值问题。这类问题在思路上通常用dp解决，但由于二叉树没法像列表那样从头到尾遍历，因此通常使用递归的方法+dp的思路，从root开始往下挖，并不断更新最值

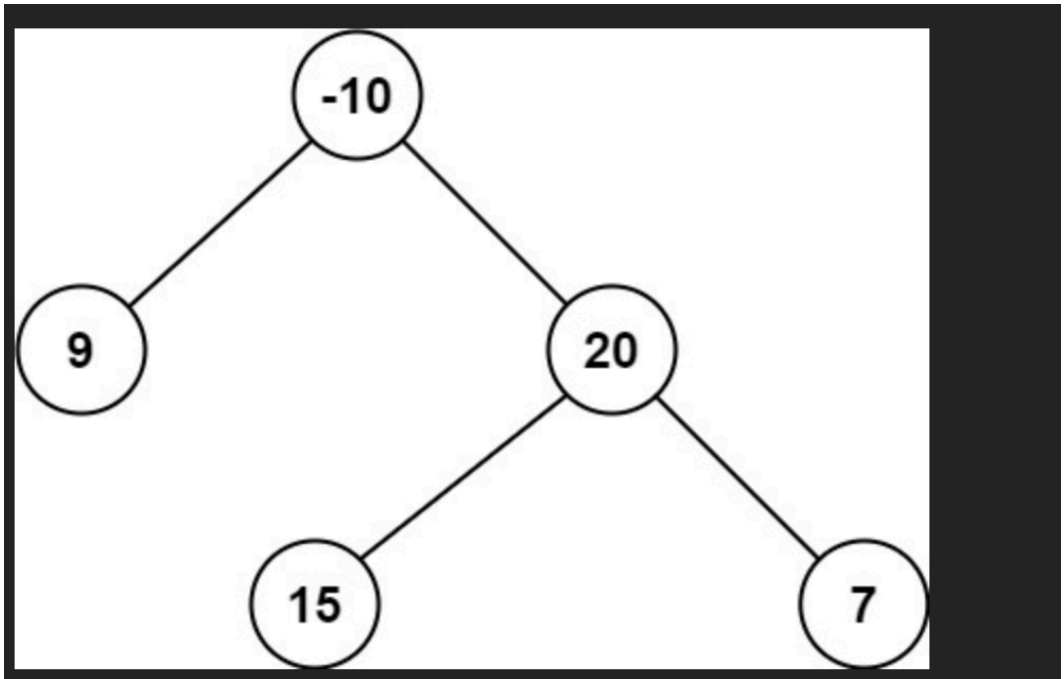
Eg1. 二叉树中的最大路径和

二叉树中的 **路径** 被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。同一个节点在一条路径序列中 **至多出现一次** 。该路径 **至少包含一个** 节点，且不一定经过根节点。

路径和 是路径中各节点值的总和。

给你一个二叉树的根节点 `root` ，返回其 **最大路径和** 。

示例：



> **输入：** root = [-10,9,20,null,null,15,7]
> **输出：** 42
> **解释：** 最优路径是 15 -> 20 -> 7 ，路径和为 15 + 20 + 7 = 42

解：其思路与动态规划题目“最大子数组和”非常类似：对于某个节点（`root`），想要找到其作为根节点的树的最大路径和，则存在4种可能：

- `root.left`和`root.right`各自作为根节点的子树中的最大路径和（且确保这两个最大路径都经过`root.left`,`root.right`）都是正数，则它们对于`root`作为根节点的树的最大路径和都是正贡献，此时经过`root`的最大路径取`left-root-right`
- 如果二者中只有一个是正的，另一个是负的，则只取正贡献的那个，最大路径取`left-root`或`right-root`
- 如果二者都是负的，则加上它们后对于根节点带来的都是负贡献，因此最大路径只取`root`节点本身

由此，可得经过每个`root`节点的最大路径和，可将其用于更新全局最大路径和`max_val`

然而，有一点需要注意的是，更新完全局最大路径值后，需要将经过当前节点的最大路径和返回并给到其父节点（也即“往上浮”），但如果这个最大路径和是`left-root-right`这种路径得到的，则其加上父节点后就会出现分叉，不能构成有效路径。因此，返回给父节点并可以为之所用的，只能是后两种情况中的最大路径和。

```python
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        max_val = [-1001] # 全局最大路径和
        def maxPathSum_(root, max_val: List):
            if not root: return 0

            max_val_left = maxPathSum_(root.left, max_val)
            max_val_right = maxPathSum_(root.right, max_val)

            # 如果不再往上追溯到话，则当前root下最大子路径包括如下4种
            # 其可以用于更新全局最大路径值
            max_val_root = max(
                root.val,
```

```
                root.val + max_val_left,
                root.val + max_val_right,
                root.val + max_val_left + max_val_right
            )
            # 更新全局最大路径值
            max_val[0] = max(max_val[0], max_val_root)

            # 而当前root还需要继续往上追溯，其返回给父节点的最大值不能是同时包括左右子树的，因为这样再加上父节点后就使得路径出现了分叉，不能构成路径了
            max_val_root_return = max(
                root.val,
                root.val + max_val_left,
                root.val + max_val_right,
            )
            return max_val_root_return
        maxPathSum_(root, max_val)

        return max_val[0]
```

Eg2. 二叉树的直径

给你一棵二叉树的根节点，返回该树的 **直径** 。

二叉树的 **直径** 是指树中任意两个节点之间**最长路径**的 **长度** 。这条路径可能经过也可能不经过根节点 `root` 。

两节点之间路径的 **长度** 由它们之间边数表示。

解：

本题为上一题“最大路径和”的低配版，也是不限制路径方向的自由路径问题：在每个节点处，同样是分别统计其左节点往下和右节点往下的最大深度，然后将左节点最大深度+当前节点贡献的1个单位+右节点最大深度 作为经过当前节点的最长路径，用于更新全局最大值；而当前节点返回给父节点的最大深度则只能够选择左/右节点下的最大深度（之一）+当前节点贡献的1个单位，以便能够和父节点连接成为有效的路径

```
class Solution:
    def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        max_path = [0]
        def depth(node, max_path: List):
            if not node: return 0

            # 左/右节点分别往下的最大深度
            max_depth_left = depth(node.left, max_path)
            max_depth_right = depth(node.right, max_path)

            # 左节点往下最大深度 + 当前节点贡献1个节点 + 右节点往下最大深度 = 经过当前节点的最长路径
            max_path_node = max_depth_left + 1 + max_depth_right
            max_path[0] = max(max_path[0], max_path_node)

            # 而当前节点返回给其父节点的结果中，只能选择左节点往下最大深度 或 右节点往下最大深度 再加上自身贡献的1个单位的深度，从而能够和父节点构成路径
            max_path_node_return = max(max_depth_left, max_depth_right) + 1
```

```
            return max_path_node_return

        depth(root, max_path)
        return max_path[0] - 1  # 最终求的是边数，为路径节点数-1
```

Eg3. 打家劫舍III

小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为 `root` 。

除了 `root` 之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果 **两个直接相连的房子在同一天晚上被打劫** ，房屋将自动报警。

给定二叉树的 `root` 。返回 ***在不触动警报的情况下***，*小偷能够盗取的最高金额* 。

解：

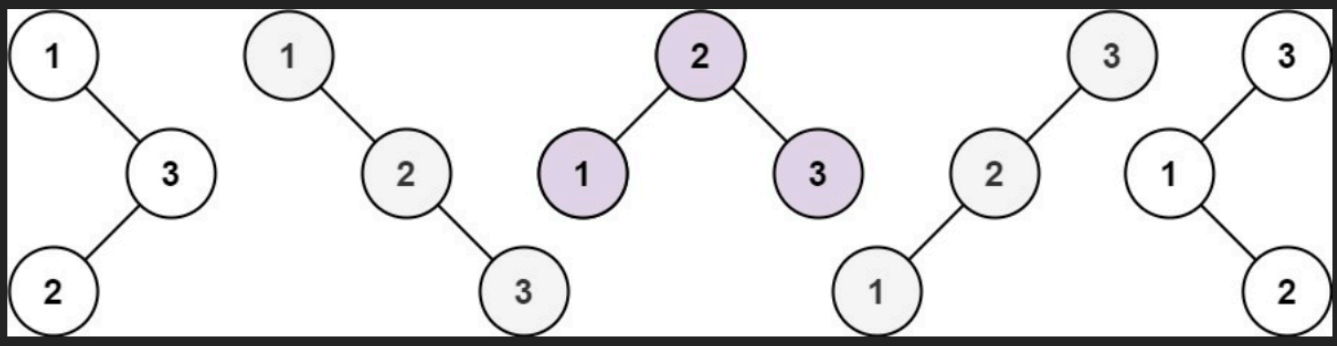
在二叉树情况下，难以使用标准的dp来遍历，因此使用递归的方式来表达传递关系。具体而言，从root开始，每遍历到一个节点，就分别对其左右节点求最值（也即，左右节点子树下分别能偷到的最大价值），并分别返回它们偷自己/不偷自己的最大价值（必须区分是否偷了左/右节点本身的结果，因为这决定了能否偷当前遍历到的节点）。然后，分别考虑偷当前节点/不偷当前节点的情况：若偷当前节点，则左右节点都只能取不偷左/右节点本身的最大价值；若不偷当前节点，则左右节点都可以随意取偷/不偷自身节点的最大价值

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def rob(self, root: Optional[TreeNode]) -> int:
        def rob_(node):
            if not node:    # 偷到底部，递归返回条件
                return (0, 0)

            # 左、右节点的子树下分别能偷到的最大值（分别考虑了是否偷左、右节点自己的情况）
            left_max_rob, left_max_no_rob = rob_(node.left)
            right_max_rob, right_max_no_rob = rob_(node.right)

            # 若偷当前节点，则不能偷其左右子节点，只能继承左右节点的no_rob情况
            node_max_rob = node.val + left_max_no_rob + right_max_no_rob
            # 若不偷当前节点，则左右节点偷/不偷的最优情况都可以选取，可以选取二者的最优情况加和
            node_max_no_rob = max(left_max_rob, left_max_no_rob) + max(right_max_rob,
right_max_no_rob)

            return node_max_rob, node_max_no_rob

        root_max_rob, root_max_no_rob = rob_(root)
        return max(root_max_rob, root_max_no_rob)
```

Eg4. 不同的二叉搜索树

给你一个整数 `n` ，求恰由 `n` 个节点组成且节点值从 `1` 到 `n` 互不相同的 **二叉搜索树** 有多少种？返回满足题意的二叉搜索树的种数。

示例 1：



> **输入：** n = 3
> **输出：** 5

解：

本题和前边几道题的递归思路不太一样，其是比较典型的dp问题。

设`dp[i]`表示总共有`i`个元素时能构造出的二叉搜索树数量，初始化为`dp[0]=1, dp[1]=1`，依次从小到大求出各个`dp[i]`直到`dp[n]`即可。

对于某个特定`i`而言，依次尝试以其范围内的第`0~i-1`个节点为根节点来构造二叉搜索树，并把每个元素作为根节点的二叉搜索树数量累加到`dp[i]`上即可。对于以节点`j`为根节点的情况，可以确定其左子树的节点为`0~j-1`，右子树的节点为`j+1~i`，也即确知左子树和右子树的元素数量，则左子树和右子树的构造是独立的（不管二者内部怎么折腾，都能保证左子树元素总是全小于j，右子树元素总是全大于j），因此以`j`为根节点的情况下能构造的二叉搜索树数量就是左子树元素数量的dp值 * 右子树元素数量的dp值。

```
class Solution:
    def numTrees(self, n: int) -> int:
        # dp[i]表示共i个节点时，可构造出的二叉搜索树数量
        dp = [0] * (n + 1)
        dp[0] = 1  # 空树
        dp[1] = 1  # 只有一个节点

        # 从小到大遍历总节点数量i
        for i in range(2, n + 1):
            # 对于0~i-1范围内的所有节点j，依次遍历以节点j为根节点时可构造出的二叉搜索树数量，每得到一个就将它加到dp[i]上
            # 由于确定以j为根节点时，左子树和右子树有哪些节点已经确定（左子树为0 ~ j-1，右子树为j+1 ~ i），因此左子树和右子树的构造是完全独立的（不管它们怎么折腾都能确保左子树全都小于根节点，右子树全都大于根节点）。
            # 因此，以j为根节点时的二叉搜索树总数就是左子树能构造出的二叉搜索树数量 * 右子树能构造出的二叉搜索树数量，这两个值可以通过左右子树元素数量对应的dp值得到
            for j in range(0, i):
```

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

        return dp[n]
```

字典树

Trie 或者说 **前缀树** 是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，例如自动补全和拼写检查。

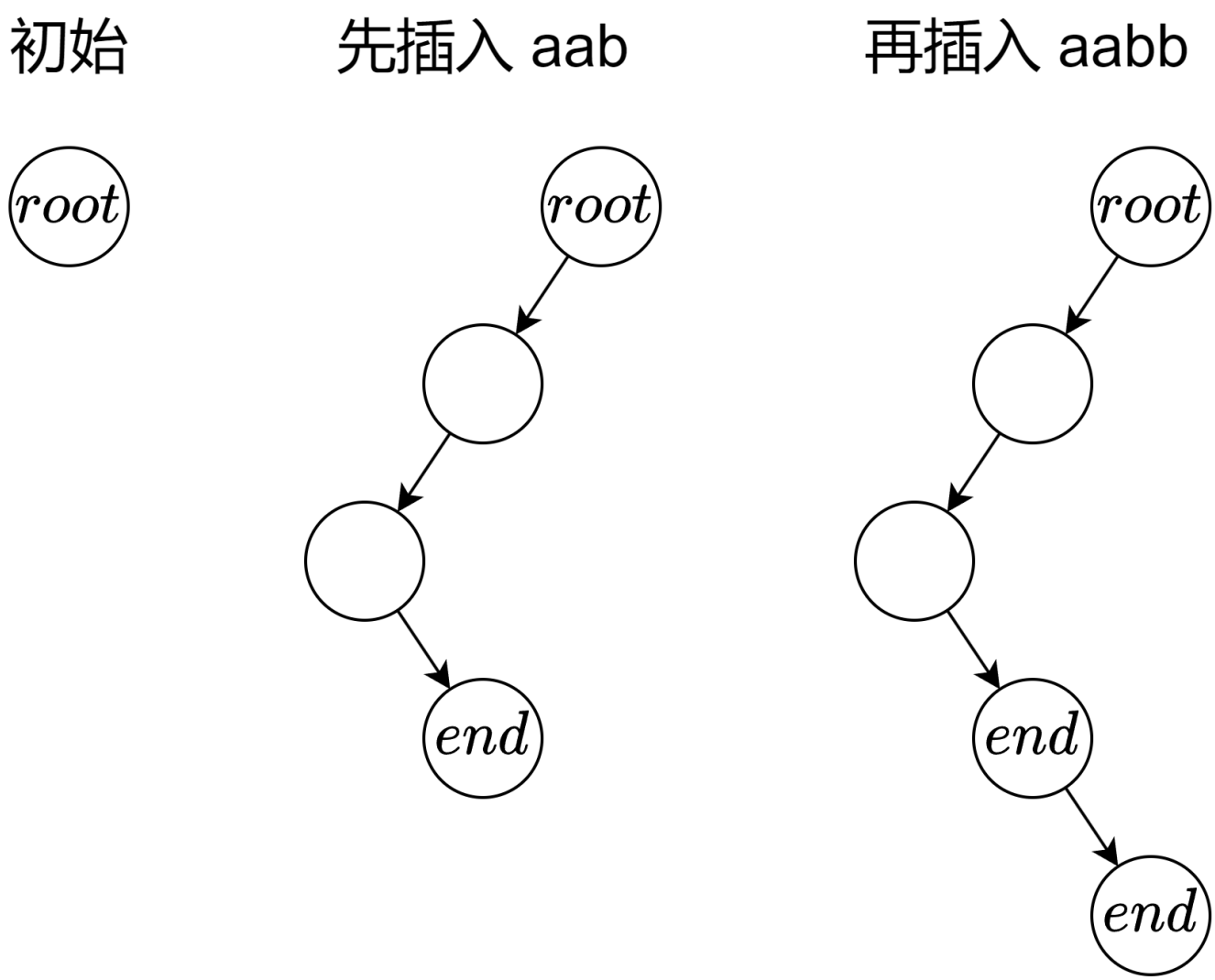
请你实现 Trie 类：

- `Trie()` 初始化前缀树对象。
- `void insert(String word)` 向前缀树中插入字符串 `word` 。
- `boolean search(String word)` 如果字符串 `word` 在前缀树中，返回 `true` （即，在检索之前已经插入）；否则，返回 `false` 。
- `boolean startsWith(String prefix)` 如果之前已经插入的字符串 `word` 的前缀之一为 `prefix` ，返回 `true` ；否则，返回 `false` 。

解：

可以把整个字典树建模为一个26叉树，初始化字典树时为其设置一个根节点。之后的每个节点代表都出现过的一个字母，其包含两个成员值：`self.sons`是一个字典，用于存储当前节点的后继节点（键:值分别为下一个字母和它对应的节点对象）；`self.has_end`表示当前是否插入过以该节点为终止的词语。这样每个插入的词都可以建模为从根节点往下走的一条路径。

例如，假设字母只有`a,b`两种，把`a`视为左子节点，把`b`视为右子节点，则：插入`"aab"`相当于生成了一条“左-左-右”的路径，并标记最后一个节点为终止节点；再插入`"aabb"`相当于生成了一条“左-左-右-右”的路径（其中前三个节点和`"aab"`是共享的），并标记最后一个节点为终止节点：



当扩展到26个字母时，相当于树变成了26叉树，但逻辑是类似的。

在查询时，只需从根结点开始，逐个匹配待查询词语的各个字母：每次都观察当前节点的`sons`字典中是否存了词语中的下一个字母对应的节点，如果有对应的话就可以继续往这个字母对应的节点走。如果某个时刻发现当前节点的子节点中没有下一个字母对应的节点，则说明该单词未存在于当前的字典树中。`search`和`startsWith`可以复用上述逻辑，区别仅在于`search`要求输入的单词的最后一个字母对应的节点一定作为过终止节点（之前插入过和当前查询的单词完全相同的单词），也即`has_end=True`，这样能够做到完整单词匹配，而`startsWith`则只需把输入的前缀匹配完即可，不需要匹配到末尾时的节点一定作为过终止节点。

```
class Node:
    def __init__(self):
        self.sons = dict()
        self.has_end = False

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

    def insert(self, word: str) -> None:
        cur = self.root
        for char in word:
            if char not in cur.sons:  # 无路可走?
```

```
                cur.sons[char] = Node()  # 那就造路!
            cur = cur.sons[char]
        cur.has_end = True

    def find(self, word: str) -> int:
        cur = self.root
        for char in word:
            if char not in cur.sons:  # 道不同，不相为谋
                return 0
            cur = cur.sons[char]
        # 走过同样的路（2=完全匹配，1=前缀匹配）
        return 2 if cur.has_end else 1

    def search(self, word: str) -> bool:
        return self.find(word) == 2

    def startsWith(self, prefix: str) -> bool:
        return self.find(prefix) != 0
```

递归与动态规划

解题框架：相当于递归的反向操作，二者都是利用的**递推关系式**来实现。

- 递归：

  从顶到底来解决问题，**将大问题不断拆解为子问题**，每个子问题都再调用一遍函数本身，直到触发边界条件而返回。

  例如，对于斐波那契数列求解：

  $$f(n) = f(n-1) + f(n-2)$$

  ```
  # 求第 n 个斐波那契数
  def fibonacci(n):
      if n == 0: return 0 # 返回 f(0)
      if n == 1: return 1 # 返回 f(1)
      return fibonacci(n - 1) + fibonacci(n - 2) # 分解为两个子问题求解
  ```

- 动态规划：

  维护一个`dp`数组，用以保存每个状态。通过初始化和设置最简单子问题的边界条件，通过组合已知的子问题的解来一步步推导得到父问题的解，从底到顶地求解问题，最终得到总问题的解。

  例如，对于斐波那契数列的求解：

  $$f(n) = f(n-1) + f(n-2)$$

```python
# 求第 n 个斐波那契数
def fibonacci(n):
    if n == 0: return 0         # 若求 f(0) 则直接返回 0
    dp = [0] * (n + 1)          # 初始化 dp 列表
    dp[0], dp[1] = 0, 1         # 初始化 f(0), f(1)
    for i in range(2, n + 1): # 状态转移求取 f(2), f(3), ..., f(n)
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]                # 返回 f(n)
```

示例2：

一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 `n` 级的台阶总共有多少种跳法。

解：

设跳上n级台阶有$f(n)$种跳法，所有跳法中，最后一步只可能跳上1级或2级，若为前者则剩余的$n-1$个台阶共有$f(n-1)$种跳法，若为后者则剩余的$n-2$个台阶共有$f(n-2)$种跳法，因此递推关系为：

$$f(n)=f(n-1)+f(n-2)$$

和斐波那契数列的唯一区别在于起始数字不同：

```python
class Solution:
    def numWays(self, n: int) -> int:
        a, b = 1, 1
        for _ in range(n):
            a, b = b, a + b
        return a
```

题型：子序列/子串问题

Eg1：最长回文子串

给定字符串 `s`，找到并返回最长回文子串（子串必须是连续的）。

解析：设置dp矩阵，取值为 `True` 或 `False`，其中 `dp[i][j]` 表示 `s[i]~s[j]` 这段子串是否是回文串。初始化时所有 `dp[i][i]` 都是 `True`，因为每个字符自己都是回文串。

这样一来就有状态转移矩阵：

$$dp[i][j]=dp[i+1][j-1]\ \&\&\ s[i]==s[j]$$

由于推导是由短到长的，因此遍历时外层循环应为子串长度 `L`，内层循环为子串起始点 `i`，这样就可以保证需要求 `dp[i][j]` 时，`dp[i+1][j-1]` 一定已经被算出来了。

另外，当 `j=i+1`，也即子串长度为2时，应单独作为边界条件设置：只要 `s[i]==s[j]` 则 `dp[i][j]=True`。

由于动态规划本身只能求得“数量统计”，并不能记录具体哪个子串是最长子串，因此还需要使用 `max_len` 和 `max_begin` 全局变量来在循环中记录当前最长子串的起始点和长度，由此可以还原出这个最长子串。

```python
class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        # 边界情况：len(s) = 1，回文串直接就是自己
        if len(s) == 1:
            return s

        # 初始条件，max_len用于保存当前见过的最大长度，max_begin用于保存最大长度的起始位置
        max_len = 1
        max_begin = 0

        # dp矩阵，dp[i][j]的bool值表示s[i]~s[j]是不是回文串
        dp = [[False for _ in range(len(s))] for _ in range(len(s))]
        # 初始化边界条件：每个字符自己都是回文串
        for i in range(len(s)):
            dp[i][i] = True

        # 从2开始遍历所有的字串长度
        for L in range(2, len(s)+1):
            # 从头开始遍历该长度下的所有字串，使用滑动窗口
            # 只需使用起始index i与长度L即可确定字串
            for i in range(len(s)):
                j = i + L - 1
                # 若字串末尾超出了字符串末尾，则直接跳出该轮循环
                if j >= len(s):
                    continue

                # 若字串长度为2，则只需看子串中的两个字符是不是相等
                if L == 2:
                    dp[i][j] = (s[i] == s[j])
                else:
                    # 若首尾字符相同，则具备了成为回文串的基本条件，此时若其字串也是回文串则其是回文串
                    if s[i] == s[j]:
                        dp[i][j] = dp[i+1][j-1]
                    else:
                        dp[i][j] = False

                # 若s[i]~s[j]这段字串为回文串，则看看其是不是目前看到最长的，若是则更新统计
                if dp[i][j] == True:
                    if L > max_len:
                        max_len = L
                        max_begin = i

        # 所有长度遍历完之后，返回此时统计到的最大字串
        return s[max_begin: max_begin + max_len]
```

Eg1.1 回文子串

给你一个字符串 `s` ，请你统计并返回这个字符串中 **回文子串** 的数目。

示例：`"abc"`中含有3个回文子串（`"a", "b", "c"`），`"aaa"`中含有6个回文子串（`"a","a","a","aa","aa","aaa"`）

解：使用`dp[i][j]`表示`s[i:j]`（左闭右闭）是否为回文子串，其递推公式为：`dp[i][j]=dp[i+1][j-1] and (s[i]==s[j])`，最终只需对dp数组求和来查看一共有多少个True的子串即可

```
class Solution:
    def countSubstrings(self, s: str) -> int:
        dp = [[False for _ in range(len(s))] for _ in range(len(s))]

        for i in range(len(s)):
            dp[i][i] = True
            if not i == len(s) - 1:
                dp[i][i+1] = (s[i] == s[i+1])

        for L in range(2, len(s)):
            for i in range(len(s)-L):
                dp[i][i+L] = dp[i+1][i+L-1] and (s[i] == s[i+L])

        return sum([sum(ls) for ls in dp])
```

Eg2. 最长递增子序列

给你一个整数数组 `nums` ，找到其中最长严格递增子序列的长度。

解析：有点类似蛋糕问题。设`dp[i]`为`nums[0]~nums[i]`之间，以`nums[i]`结尾的最长严格递增子序列的长度。在计算每个`dp[i]`时，状态转移为：

$$
\begin{gathered}
dp[i] = \max(dp[j]) + 1, \\
\text{其中：} \\
0 \le j < i, \\
num[j] < num[i]
\end{gathered}
$$

也即，在`0~i`之间依次遍历所有`j`，若`nums[j]<nums[i]`，则说明`nums[0]~nums[j]`之间任何以`nums[j]`结尾的递增子序列加上`nums[i]`后依然是递增子序列。因此，遍历所有`j`，并找到`nums[j]<nums[i]`的这些`j`中`dp[j]`最大的，其加上1就是`dp[i]`的值。

```
class Solution(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0

        # dp[i] 表示以nums[i]结尾的最长递增子序列长度，必包括nums[i]
        dp = [1 for _ in range(len(nums))]
```

```
        for i in range(1, len(nums)):
            max_subseq = 1
            for j in range(0, i):
                if nums[j] < nums[i]:
                    max_subseq = max(max_subseq, dp[j] + 1)
            dp[i] = max_subseq

        return max(dp)
```

Eg2.2 跳跃游戏II

给定一个长度为 `n` 的 0 索引整数数组 `nums`。初始位置在下标 0。

每个元素 `nums[i]` 表示从索引 `i` 向后跳转的最大长度。换句话说，如果你在索引 `i` 处，你可以跳转到任意 `(i + j)` 处：

- `0 <= j <= nums[i]` 且
- `i + j < n`

返回到达 `n - 1` 的最小跳跃次数。测试用例保证可以到达 `n - 1`。

示例：

```
输入：nums = [2,3,1,1,4]
输出：2
解释：跳到最后一个位置的最小跳跃数是 2。
     从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
```

解：

本题的解法和最长递增子序列基本一致，也是类似蛋糕问题。设`dp[i]`表示从初始位置跳到第`i`个位置的最少步数，则对于每个`dp[i]`，依次遍历`j=0~i-1`，若在`j`位置能一步跳到`i`（`j+nums[j]>=i`），则`dp[j]+1`为经过`j`跳到`i`的最小步数，对于每个符合条件的`j`取其中最小的即为`dp[i]`的值

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        if len(nums) == 1: return 0
        n = len(nums)

        dp = [1e5] * n      # dp[i]：跳到i位置的最小步数
        dp[0] = 0

        for i in range(1, n):
            for j in range(0, i):
                if j + nums[j] >= i:
                    dp[i] = min(dp[i], dp[j]+1)

        return dp[n-1]
```

Eg3. 最长回文子序列

给定字符串，返回其最长回文子序列的长度。子序列可以是不连续的，如`[1,3,5]`是`[1,2,3,4,5,6]`的一个子序列。

解：

设`dp[i][j]`为`s[i]~s[j]`之间最长回文子序列的长度，则：

- 若`s[i]==s[j]`，则`s[i],s[j]`可以被同时添加并为最长回文子序列做出贡献，可以在`dp[i+1][j-1]`最大回文长度的基础上再+2，也即`dp[i][j] = 2 + dp[i+1][j-1]`
- 若`s[i]!=s[j]`，则`s[i],s[j]`无法同时为最长回文子序列做出贡献，只能从分别添加`s[i]`或`s[j]`二者之一的情况中选一个最大的回文长度来继承（因为二者同时添加并不会带来额外收益），也即`dp[i][j] = max(dp[i+1][j], dp[i][j-1])`

注意，由于计算`dp[i][j]`时需要`dp[i+1]`，因此`i`应从后往前推，`j`应从`i`往后推：

```
class Solution(object):
    def longestPalindromeSubseq(self, s):
        """
        :type s: str
        :rtype: int
        """
        # dp[i][j]表示[i,j]范围内的最长回文子序列长度
        dp = [[0] * len(s) for _ in range(len(s))]
        # 初始化：dp[i][i] = 1
        for i in range(len(s)):
            dp[i][i] = 1

        # 遍历，由于dp[i][j] = max(dp[i+1][j], dp[i], [j-1]) if s[i] != s[j] else dp[i+1][j-1]+2，因此i需要从后往前推，j需要从前往后推
        for i in range(len(s)-1, -1, -1):
            for j in range(i+1, len(s)):
                if s[i] == s[j]:
                    dp[i][j] = dp[i+1][j-1] + 2
                else:
                    dp[i][j] = max(dp[i][j-1], dp[i+1][j])

        return dp[0][len(s)-1]
```

Eg4: 最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列** ，返回 `0` 。

解：

二维动态规划问题。设`dp[i][j]`表示`text1[0:i]`与`text2[0:j]`的最长公共子序列长度（左闭右闭），考虑二者末尾`text1[i]`和`text2[j]`是否相等：

若`text1[i]==test2[j]`，则`dp[i][j] = dp[i-1][j-1] + 1`，也即先找到`text1[0:i-1]`和`text2[0:j-1]`的最大匹配长度`dp[i-1][j-1]`，然后在其基础上再加上`text1[i],text2[j]`匹配带来的+1长度增益；

否则，由于`text1[i]`和`text2[j]`匹配不上，因此只能从退一步的结果里选一个最大的来继承：`dp[i][j] = max(dp[i-1][j], dp[i][j-1])`，与上文最长回文子序列类似。

边界条件为`i==0`或`j==0`时，单独讨论即可

```
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        dp = [[0] * len(text2) for _ in range(len(text1))]

        for i in range(0, len(text1)):
            for j in range(0, len(text2)):
                # 若text1[i] == text2[j]
                if text1[i] == text2[j]:
                    # 边界情况：i或j为首项，则公共子序列长度为1
                    if i == 0 or j == 0:
                        dp[i][j] = 1
                    else:
                        dp[i][j] = dp[i-1][j-1] + 1
                # 若text1[i] != text2[j]
                else:
                    # 边界情况：若i和j都是首项，且nums[i]!=nums[j]，则公共子序列长度为0
                    if i == 0 and j == 0:
                        dp[i][j] = 0
                    # 边界情况，若i为首项但j不是，则dp[i][j]=dp[i][j-1]
                    elif i == 0:
                        dp[i][j] = dp[i][j-1]
                    elif j == 0:
                        dp[i][j] = dp[i-1][j]
                    else:
                        dp[i][j] = max(dp[i-1][j], dp[i][j-1])

        return dp[len(text1)-1][len(text2)-1]
```

Eg5. 不同的子序列

给你两个字符串 `s` 和 `t` ，统计并返回在 `s` 的 **子序列** 中 `t` 出现的个数。

示例：

```
输入：s = "rabbbit", t = "rabbit"
输出：3
解释：
如下所示, 有 3 种可以从 s 中得到 "rabbit" 的方案。
(rabb)b(it)
(ra)b(bbit)
(rab)b(bit)
```

解：

仍使用二维动态规划：`dp[i][j]`表示`s[0:i]`的子序列中出现`t[0:j]`的次数（左闭右开）。则面对`dp[i][j]`时，考虑`s[0:i]`和`t[0:j]`各自的末尾元素`s[i-1]`和`t[j-1]`：

- 若`s[i-1]==t[j-1]`，则可以选择：1）让`s[i-1]`和`t[j-1]`进行匹配，此时子序列数目等于`dp[i-1][j-1]`（也即`s[0:i-1]`中含有的`t[0:j-1]`数目，最后再分别在后边加上`s[i-1],t[j-1]`）；2）也可以选择不让`s[i-1]`和`t[j-1]`匹配，此时子序列数目等于`dp[i-1][j]`（也即`s[0:i-1]`中含有`t[0:j]`的数目，不管`s[i-1]`）

  两种情况之和即为总的数量：`dp[i][j]=dp[i-1][j-1]+dp[i-1][j]`
- 若`s[i-1]!=t[j-1]`，则`s[i-1]`不能匹配`t[j-1]`，也即添加了`s[i-1]`后没有任何作用，只能退一步，继承`s[0:i-1]`中含有`t[0:j]`的数量，也即`dp[i-1][j]`的情况。

  此时：`dp[i][j]=dp[i-1][j]`

初始化时即考虑`s[0:i],t[0:j]`为空序列的情况

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        if len(t) > len(s): return 0

        # dp[i][j]表示s[0:i]的子序列中含有的t[0:j]数量（左闭右开）
        dp = [[0 for _ in range(len(t)+1)] for __ in range(len(s)+1)]

        # 初始化：dp[i][0] = 1，因为空串t是任何s的子序列；dp[0][j] = 0 for j != 0，因为若s为空串的话则其不可能含有非空串的t
        for i in range(len(s)+1):
            dp[i][0] = 1

        # 状态转移：
        # 考虑dp[i][j]，对于s[0:i]和t[0:j]的最后一个字符s[i-1],t[j-1]:
        # 若s[i-1]==t[j-1]，则可以选择让s[i-1]和t[j-1]进行匹配，此时子序列数目等于dp[i-1][j-1]（也即s[0:i-1]中含有的t[0:j-1]数目，最后再分别在后边加上s[i-1],t[j-1]）；也可以选择不让s[i-1]和t[j-1]匹配，此时子序列数目等于dp[i-1][j]（也即s[0:i-1]中含有t[0:j]的数目，不管s[i-1]）；两种情况之和即为总的数量
        # 若s[i-1]!=t[j-1]，则s[i-1]不能匹配t[j-1]，只能继承s[0:i-1]中含有t[0:j]的数量，也即dp[i-1][j]的情况
        for i in range(1, len(s)+1):
            for j in range(1, len(t)+1):
                if s[i-1] == t[j-1]:
                    dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
                else:
                    dp[i][j] = dp[i-1][j]
```

```
        return dp[len(s)][len(t)]
```

题型：区间dp

> 此类题通常无法像普通dp那样从头到尾推导更新，也即状态之间的依赖关系并不是前后依赖的，而是大区间状态依赖小区间。此时，通常定义`dp[i][j]`为区间`i~j`之间的状态，其依赖于`i~j`之间某些子区间的状态。dp时从最小长度的区间初始化，并逐步外推长度。
>
> “最长回文子串”问题也是典型的区间dp问题，详见相关题目

Eg1. 戳气球

有 `n` 个气球，编号为`0` 到 `n - 1`，每个气球上都标有一个数字，这些数字存在数组 `nums` 中。

现在要求你戳破所有的气球。戳破第 `i` 个气球，你可以获得 `nums[i - 1] * nums[i] * nums[i + 1]` 枚硬币。 这里的 `i - 1` 和 `i + 1` 代表和 `i` 相邻的两个气球的序号。如果 `i - 1`或 `i + 1` 超出了数组的边界，那么就当它是一个数字为 `1` 的气球。

求所能获得硬币的最大数量。

示例：

```
输入：nums = [3,1,5,8]
输出：167
解释：
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins =  3*1*5    +   3*5*8   +  1*3*8  + 1*8*1 = 167
```

解：

该问题属于区间dp问题：状态依赖并不是从前到后的，而是从小区间到大区间的，因此状态定义也要从区间出发。为了方便起见，分别添加虚拟的左右边界（来处理戳破原数组边界气球`nums[0]`和`nums[n-1]`的情况）。然后定义状态：`dp[i][j]`表示戳破`nums[i] ~ nums[j]`区间（不包括i,j）内所有气球能获得的最大硬币数量。考虑区间内最后戳破的某个气球k，则戳破它之前能获得的最大收益为其左右两个子区间的最大收益：`dp[i][k]+dp[k][j]`，最后戳破这个k时获得`nums[i]*nums[k]*nums[j]`。因此，`dp[i][j]`的状态转移即为：遍历`i~j`之间的所有`k`，找到能使得`dp[i][k] + nums[i]*nums[k]*nums[j] + dp[k][j]`最大化的情况，即可推得`dp[i][j]`。

从最短长度1开始遍历，逐步扩大区间：

```python
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)
        # 添加虚拟左右边界，此时nums长度变为n+2，nums[1]~nums[n]为原数组
        nums = [1] + nums + [1]
```

```
        # dp[i][j]表示戳破nums[i] ~ nums[j]区间（不包括i,j）内所有气球能获得的最大硬币数量
        # 最后一个戳破的气球是某个i<k<j,将其戳破时获得nums[i]*nums[k]*nums[j]
        # 初始化：所有dp[i][i+1] = 0
        # 转移关系为：dp[i][j] = max_k (dp[i][k] + nums[i]*nums[k]*nums[j] + dp[k][j])
        dp = [[0] * (n+2) for _ in range(n+2)]

        # 从小到大遍历区间长度，区间长度为nums[i] ~ nums[j]区间内部的气球数量
        for length in range(1, n+1):
            # 遍历左端点i
            for i in range(0, n+1-length):
                # 由左端点和区间长度得到右端点j
                j = i + length + 1
                # 遍历中间点k，找到能使得dp[i][j]最大的k
                for k in range(i+1, j):
                    dp[i][j] = max(dp[i][j], dp[i][k] + nums[i]*nums[k]*nums[j] + dp[k]
[j])
        return dp[0][n+1]
```

题型：二维动态规划，正方形/三角形路径数量问题

Eg1. 正方形中的不同路径

一个机器人位于一个 `m x n` 网格的左上角

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角。

问总共有多少条不同的路径?

解析：设置二维dp数组，`dp[i][j]`表示从起始点到`(i,j)`点的路径数量。由于每个点只能从左侧或上边进入，因此转移方程：`dp[i][j] = max(dp[i-1][j], dp[i][j-1])`。最终，返回右下角（终点）的dp值即可。

注意先设好边界条件：一般也即图的边界，这里任何`dp[i][0]`和`dp[0][j]`都为1，因为左边和上边均只有一条路可达。

```
class Solution(object):
    def uniquePaths(self, m, n):
        """
        :type m: int
        :type n: int
        :rtype: int
        """
        # 对于矩阵类的，注意先把左边和上边的边界条件都预定义好，而不仅仅是初始点，否则容易出错
        f = [[1] * n] + [([1] + [0] * (n-1)) for _ in range(m-1)]
        for i in range(1,m):
            for j in range(1, n):
                f[i][j] = f[i-1][j] + f[i][j-1]
        return f[m-1][n-1]
```

Eg2：三角形最小路径和

给定一个三角形 `triangle` ，找出自顶向下的最小路径和。

每一步只能移动到下一行中相邻的结点上。**相邻的结点** 在这里指的是 **下标** 与 **上一层结点下标** 相同或者等于 **上一层结点下标 + 1** 的两个结点。也就是说，如果正位于当前行的下标 `i` ，那么下一步可以移动到下一行的下标 `i` 或 `i + 1` 。

解析：类似上题，设计`dp[i][j]`作为到达`(i,j)`点的最小路径和。注意每轮循环中三角形两个侧边上为边界条件：其只能顺着侧边推下来。最后只需统计三角形底部这一行的最小dp值即可。

```
class Solution(object):
    def minimumTotal(self, triangle):
        """
        :type triangle: List[List[int]]
        :rtype: int
        """
        num_levels = len(triangle)
        if num_levels == 1:
            return triangle[0][0]
        res = [[0] * (i+1) for i in range(num_levels)]
        res[0][0] = triangle[0][0]
        for level in range(1, num_levels):
            # 两条侧边的边界条件
            res[level][0] = res[level-1][0] + triangle[level][0]
            res[level][level] = res[level-1][level-1] + triangle[level][level]
            # 其余的
            for i in range(1, level):
                res[level][i] = min(res[level-1][i-1], res[level-1][i]) + triangle[level]
[i]
        return min(res[num_levels-1])
```

Eg3. 最大正方形

在一个由 `'0'` 和 `'1'` 组成的二维矩阵内，找到只包含 `'1'` 的最大正方形，并返回其面积。

解析：设`dp[i][j]`为以`(i,j)`为右下角，且只包含1的正方形边长的最大值。如果`(i,j)`点的值本身为0则不可能存在以该点为右下角点正方形，`dp[i][j]=0`；否则`dp[i][j]`的值由其上方、左侧、左上方的三个点的dp值决定：

$$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-1], dp[i][j-1]) + 1$$

边界条件：对于左边和侧边，也即`(i,0)`和`(0,j)`，则该点本身为1时`dp=1`，否则`dp=0`

```
class Solution(object):
    def maximalSquare(self, matrix):
        """
        :type matrix: List[List[str]]
        :rtype: int
```

```
        """
        m, n = len(matrix), len(matrix[0])
        res = [[0 for _ in range(n)] for _ in range(m)]
        # 上边边界条件
        res[0] = map(lambda x:int(x), matrix[0])
        for i in range(1, m):
            # 左侧边边界条件
            res[i][0] = 1 if matrix[i][0] == '1' else 0
            # 其他
            for j in range(1, n):
                res[i][j] = min(res[i-1][j], res[i-1][j-1], res[i][j-1]) + 1 if matrix[i]
[j] == '1' else 0
        return max(max(row) for row in res) ** 2
```

题型：蛋糕/背包问题

蛋糕问题

经典情境：

针对不同重量的蛋糕设定了不同的售价，例如：

蛋糕重量	0	1	2	3	4	5	6
售价	0	2	3	6	7	11	15

现给定一个重量为n的蛋糕，问该如何切分蛋糕，来达到最高的蛋糕总售价。

解：

对于重量为n的蛋糕，其最高总售价只可能由n种组合得到：重量为$0, 1, 2, \cdots, n-1$的蛋糕的**最高售价**加上重量为$n, n-1, \cdots, 1$的蛋糕的**单独售价**，本题的解$f(n)$即为其中售价最高的一种组合。

设蛋糕售价列表为p，其中$p(i)$表示单独出售重量为i的蛋糕的价格，则本题的递推式为：

$$f(n) = \max_{0 \le i < n} (f(i) + p(n-i))$$

这里`dp[i]`即为蛋糕总重量为`i`时，能卖到的最高价格

```
# 输入蛋糕价格列表 price_list ，求重量为 n 蛋糕的最高售价
def max_cake_price(n, price_list):
    if n <= 1: return price_list[n] # 蛋糕重量 <= 1 时直接返回
    dp = [0] * (n + 1)              # 初始化 dp 列表，dp[i]为蛋糕总重量为i时能够卖出的最高价格
    for j in range(1, n + 1):       # 按顺序计算 f(1), f(2), ..., f(n)
        for i in range(j):          # 从 j 种组合种选择最高售价的组合作为 f(j)
            dp[j] = max(dp[j], dp[i] + price_list[j - i])
    return dp[n]
```

蛋糕问题的特点：用于拼接/组合/累加的各种可选的“子部分”都是无限的。

在上述例题中，“一种蛋糕重量”即为“用于累加的子部分”，并且每种蛋糕重量都是可以无限使用的（例如，总重量为3的蛋糕可以切成3个重量为1的，重复使用多个重量为1的子部分完全没问题）

在这种情形下，设`dp[i]`为拼接/组合/累加得到`i`时的状态，在外层循环中遍历`1~target`，内层循环可选的“子部分”

Eg 1. 单词拆分

给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 `s` 则返回 `true`。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

示例：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以由 "apple" "pen" "apple" 拼接成。
```

解：本质上和蛋糕问题/最长递增子序列类似（**因为每个单词都可以无限使用**）：使用1D dp，`dp[i]`表示`s[0:i]`能否被成功完全用集合中的字串进行切分。这样一来，对于`dp[i]`，遍历`dp[0]~dp[i-1]`，当某个`dp[j]==True`时，查看`s[j:i]`是否在集合中，如果是则`dp[i]=True`，如果最终都没有发现这种关系的话则`dp[i]=False`

```python
class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: List[str]
        :rtype: bool
        """

        dp = [False for _ in range(len(s)+1)]
        dp[0] = True

        for i in range(1, len(s) + 1):
            for j in range(0, i):
                if dp[j] and (s[j:i] in wordDict):
                    dp[i] = True
                    break
        return dp[len(s)]
```

Eg2. 零钱兑换

给你一个整数数组 `coins` ，表示不同面额的硬币；以及一个整数 `amount` ，表示总金额。

计算并返回可以凑成总金额所需的 **最少的硬币个数** 。如果没有任何一种硬币组合能组成总金额，返回 `-1` 。

你可以认为每种硬币的数量是无限的。

示例：

```
输入：coins = [1, 2, 5], amount = 11
输出：3
解释：11 = 5 + 5 + 1
```

解：

本题同样为蛋糕问题（**因为每种金额的硬币都可以无限使用**）：设`dp[i]`为总金额为`i`时的最少硬币个数，则其状态转移为：遍历所有可能存在的硬币面额，对于面额`coin_val`，其本身给`dp[i]`贡献了1枚硬币，则`dp[i]`可以由`dp[i-coin_val] + 1`转化而来，遍历所有硬币面额后取值最小的情况即可。

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [10 ** 5] * (amount+1)
        dp[0] = 0

        for i in range(amount+1):
            for coin_val in coins:
                if coin_val <= i:
                    dp[i] = min(dp[i], dp[i-coin_val] + 1)

        return dp[amount] if dp[amount] != 10 ** 5 else -1
```

Eg3. 完全平方数

给你一个整数 `n` ，返回 *和为 `n` 的完全平方数的最少数量* 。

完全平方数是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，`1`、`4`、`9` 和 `16` 都是完全平方数，而 `3` 和 `11` 不是。

示例 ：

```
输入：n = 12
输出：3
解释：12 = 4 + 4 + 4

输入：n = 13
输出：2
解释：13 = 4 + 9
```

解：该问题为蛋糕问题，因为目标整数拆成的完全平方数中，可以无限重复使用同一个完全平方数。设`dp[i]`为最少需要多少个完全平方数来求和得到目标整数`i`，则从1开始从小到大遍历`i`，对于每个`dp[i]`来说，依次查看`dp[i-1],dp[i-4],dp[i-9],...`的值，它们中的最小值+1（也即其加上1个完全平方数后能得到`i`）就是`dp[i]`的值

```
class Solution:
    from math import sqrt
    def numSquares(self, n: int) -> int:
        # dp[i] 表示最少需要多少个平方数来表示整数i，最后要求的就是dp[n]
        dp = [0] * (n + 1)
        dp[1] = 1

        # 对于dp[i]，从1开始遍历：逐个尝试dp[i-1], dp[i-4], dp[i-9], ..., 直到i-sqrt(i)，则
dp[i] = min(dp[i-1], dp[i-4], dp[i-9], ...) + 1
        for i in range(2, n + 1):
            min_val = 10**5
            for j in range(1, int(sqrt(i))+1):
                min_val = min(min_val, dp[i-j**2])
            dp[i] = min_val + 1
        return dp[n]
```

Eg4. 分割回文串II

给你一个字符串 `s`，请你将 `s` 分割成一些子串，使每个子串都是回文串。

返回符合要求的 **最少分割次数** 。

示例：

```
输入：s = "aab"
输出：1
解释：只需一次分割就可将 s 分割成 ["aa","b"] 这样两个回文子串。
```

解：

本题是回文串+蛋糕问题的结合。直接用`dp[i][j]`表示`s[i]~s[j]`最少能由多少回文子串拼成不太容易，因此可以拆解成两步：首先使用回文串的推导方式来得到所有可能的子串`s[i]~s[j]`是否为回文串（`is_palindrome[i][j]`），然后再用一个一维的`dp[j]`表示`s[0]~s[j]`最少可以由多少个回文子串拼成，该步可以用蛋糕问题的方法解决：遍历所有`j in (0, i)`，若`is_h[i][j] == True`则可以刷新`dp[j] = min(dp[j], dp[i-1] + 1)`：

```
class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)

        # 1. 预处理，构建回文判断矩阵
        is_palindrome = [[False] * n for _ in range(n)]

        # 填充回文矩阵
        for i in range(n):
```

```
            is_palindrome[i][i] = True

        for length in range(2, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                if length == 2:
                    is_palindrome[i][j] = (s[i] == s[j])
                else:
                    is_palindrome[i][j] = (s[i] == s[j]) and is_palindrome[i + 1][j - 1]

        # 2. 动态规划计算最小分割次数
        dp = [0] * n

        for i in range(n):
            # 最坏情况：每个字符都分割一次
            min_cuts = i

            for j in range(i + 1):
                # 如果 s[j:i+1] 是回文
                if is_palindrome[j][i]:
                    if j == 0:
                        # s[0:i+1] 整个是回文，不需要分割
                        min_cuts = 0
                    else:
                        # 在 j-1 处分割，加上一次分割
                        min_cuts = min(min_cuts, dp[j - 1] + 1)

            dp[i] = min_cuts

        return dp[n - 1]
```

背包问题

经典情境：

给定 `n` 个物品，每个物品有一个重量 `weight[i]` 和一个价值 `value[i]`。现在有一个容量为 `W` 的背包，问如何选择物品放入背包，使得背包中物品的总价值最大？请给出背包能装下的最大价值。注意：每个物品只能选择一次（要么放入，要么不放入）。

解：令`dp[j]`表示背包容量为`i`时能装下的最大价值。

若使用蛋糕问题的思路：

```python
def knapsack_wrong(weight, value, W):
    n = len(weight)
    dp = [0] * (W + 1)  # dp[j]为背包容量为j时能装下的最大价值

    # 外层：从小到大遍历 1 ~ target
    for i in range(1, W+1):
        # 内层：遍历各个物品：weight[0] ~ weight[n-1]
        for j in range(n):
            dp[i] = max(dp[i - weight[j]] + value[j], dp[i])
```

这样会出现错误：设`m<n`，则可能出现情况：`dp[m]`的结果中已经用过了某个物品`(weight[k], value[i])`，而到计算`dp[n]`时由于物品还是从物品列表开头重新开始遍历到，所以还会遍历到`k`物品，这就导致存在物品被重复使用的风险！

若调换内外层遍历顺序：

```python
def knapsack_wrong(weight, value, W):
    n = len(weight)
    dp = [0] * (W + 1)  # dp[j]为背包容量为j时能装下的最大价值

    # 外层：遍历各个物品：weight[0] ~ weight[n-1]
    for j in range(n):
      # 内层：从小到大遍历 1 ~ target
      for i in range(1, W+1):
            dp[i] = max(dp[i - weight[j]] + value[j], dp[i])
```

这样仍然存在问题：虽然在外层逐个遍历物品可以在一定程度上避免物品重用的风险，但是在每轮外循环内部遍历更新`dp[1]~dp[target]`的过程中，由于是从小到大遍历更新的，而在当前物品`j`存在下`dp[i]`的更新又依赖于在其之前更新的`dp[i-weight[j]]`，有可能`dp[i-weight[j]]`更新的结果本身已经用过了`j`物品，此时`dp[i]`再更新时就会重复使用`j`物品！

因此，正确方案为：外层遍历各个物品，内层**从大到小**反向遍历target来更新dp，避免重复计算

```python
def knapsack(weight, value, W):
    n = len(weight)
    dp = [0] * (W + 1)  # dp[j]为背包容量为j时能装下的最大价值

    # 外层：遍历各个物品 weight[0] ~ weight[n-1]
    for i in range(n):
        # 内层：从最终目标开始，倒序遍历target
        for j in range(W, weight[i] - 1, -1):  # 反向遍历防止重复计算
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i])

    return dp[W]
```

背包问题的特点：用于拼接/组合/累加的各种可选的“子部分”都是有限的。

在这种情形下，设`dp[i]`为拼接/组合/累加得到`i`时的状态，在外层循环中遍历各个“子部分”，内层循环从大到小反向遍历更新target

Eg1. 分割等和子集

给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

示例 1：

```
输入：nums = [1,5,11,5]
输出：true
解释：数组可以分割成 [1, 5, 5] 和 [11] 。
```

示例 2：

```
输入：nums = [1,2,3,5]
输出：false
解释：数组不能分割成两个元素和相等的子集。
```

解：只需转化为总容量`target=sum(nums)/2`的背包问题即可，`dp[i]`表示是否存在能够凑成和为`i`的子集

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        sum_all = sum(nums)
        if sum_all % 2 != 0: return False
        tgt = int(sum_all / 2)

        # dp[i]表示是否存在能够凑成和为i的子集
        dp = [False] * (tgt+1)
        dp[0] = True    # 空集可以凑成0

        for num in nums:
            for i in range(tgt, -1, -1):
                if num <= i:
                    if dp[i-num]:
                        dp[i] = True
        return dp[tgt]
```

Eg2. 目标和

给你一个非负整数数组 `nums` 和一个整数 `target` 。

向数组中的每个整数前添加 `'+'` 或 `'-'` ，然后串联起所有整数，可以构造一个 **表达式** ：

- 例如，`nums = [2, 1]`，可以在 `2` 之前添加 `'+'`，在 `1` 之前添加 `'-'`，然后串联起来得到表达式 `"+2-1"`。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同 **表达式** 的数目。

示例：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 3 。
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

解：

首先也将其转化为背包问题。设取正号的元素集合为P，取负号的元素集合为Q，则有：
$sum(P) - sum(Q) = target$

另外，还知道这些元素的总和（也即原始列表元素总和）：$sum(P) + sum(Q) = total = sum(nums)$

求解可得：$sum(P) = (target + total)/2$，$sum(Q) = (total - target)/2$

因此，问题转化成：从`nums`中找出一些元素构成集合P，使得其和为$sum(P) = (target + total)/2$，问有多少种取值组合（或者考虑$Q$也可以，都是等价的）

设`dp[i]`表示和为`i`的组合数量，`dp[0]=1`为边界条件表示tgt=0时只有一个空集。另外，需要保证$sum(P), sum(Q)$都必须是整数且大于0，否则不可能存在满足条件的组合。

状态转移为：对于数组中每个元素`num`，对于某个目标值`tg`，`num`为其贡献的组合数量为`dp[tg-num]`（若`num<=tgt`），因此有`dp[tg] += dp[tg-num]`。这里需要注意的是：1）循环外层必须是`num`，内层是目标值`tg`，因为如果反过来（外层`tg`，内层`num`）的话，会导致同一个数字被多次使用；2）`tg`的循环必须是从大到小的，如果从小到大也会导致num重复使用。

```python
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:

        total = sum(nums)
        if (total + target) % 2 != 0: return 0
        if (total - target) / 2 < 0 or (total + target) / 2 < 0: return 0

        # 问题转化成：寻找和为tgt的组合数量
        tgt = int((total + target) / 2)
        # dp[i]表示和为i的组合数量
        dp = [0] * (tgt + 1)
        dp[0] = 1   # 和为0的组合只有空集
        for num in nums:    # 遍历nums中的各个数字
            # 从大到小更新dp[tg]
            for tg in range(tgt, num-1, -1):
                dp[tg] += dp[tg-num]
        return dp[tgt]
```

题型：最大子数组和/积问题

由于数组元素可能为正也可能为负，因此需要考虑负贡献情况，如果是负贡献的话那就不如直接取自身。

Eg1：最大子数组和

给你一个整数数组 `nums` ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素，且必须是原数组中连续的一部分），返回其最大和

示例：

```
输入：nums = [-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 。
```

解：

设`dp[i]`表示以`nums[i]`结尾的最大和子数组，则其转移关系为：`dp[i]=max(dp[i-1]+nums[i], nums[i])`，这是因为：`dp[i-1]`如果是正数的话，那么`dp[i-1]+nums[i]`就是以`nums[i]`结尾的最大和子数组；但如果`dp[i-1]`本身为负数的话，那么`dp[i-1]+nums[i]`会比`nums[i]`本身还小，也即拉了`nums[i]`的后腿，此时以`nums[i]`结尾的最小大子数组就是`nums[i]`本身构成的单元素子数组。最后只需挑出最大的`dp[i]`即为整个数组中和最大的子数组

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        if len(nums) == 1: return nums[0]
        dp = [0] * len(nums)

        dp[0] = nums[0]
        for i in range(1, len(nums)):
            dp[i] = max(dp[i-1] + nums[i], nums[i])

        return max(dp)
```

Eg2：乘积最大子数组

给你一个整数数组 `nums` ，请你找出数组中乘积最大的非空连续 子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

示例：

```
输入: nums = [2,3,-2,4]
输出: 6
解释: 子数组 [2,3] 有最大乘积 6。
```

解：

本题也使用动态规划的思路来解决：从头到尾依次求出以每个数组元素结尾的乘积最大子数组，然后再挑出其中的最大值即可。

然而，问题在于本题不能简单地通过前一个元素结尾的乘积最大子数组乘积值（`dp[i-1]`）来推出`dp[i]`，因为存在负数的情况。假设递推式为`dp[i]=max(dp[i-1]*nums[i], nums[i])`，也即前一个元素的dp值乘以`nums[i]`或`nums[i]`本身，则例如`nums=[5,6,-3,4,-3]`时，有`dp=[5, 30, -3, 4, -3]`，但事实上到最后一个`-3`时由于正负数相消因此最后一个dp值应该是`5*6*(-3)*4*(-3)`。

因此，考虑到正负数的问题，可以发现：若`nums[i]`为正数，则希望存在一个正的很大的`dp[i-1]`（这里代指以`nums[i-1]`为结尾的某个子数组乘积），这样`nums[i]*dp[i-1]`就会很大；若`nums[i]`为负数，则希望存在一个负的很小（绝对值很大）的`dp[i-1]`，这样`nums[i]*dp[i-1]`也会很大；如果无论是最大的`dp[i-1]`还是最小的`dp[i-1]`都和`nums[i]`符号相反，则它们和`nums[i]`相乘都会拉低结果，此时直接取`nums[i]`本身即可。

由此可见，对于每个`nums[i]`可以维护两个dp数组：`dp_max[i]`表示以`nums[i]`为结尾的乘积最大的子数组，`dp_min[i]`表示以`nums[i]`为结尾的乘积最小的子数组，而它们各自分别都可能是由`dp_max[i-1]`或`dp_min[i-1]`得到（也即`dp_max[i]`不一定推导自`dp_max[i-1]`，也可能来自`dp_min[i-1]`，反之亦然），可得转移关系：

```
dp_max[i] = max(dp_max[i-1] * nums[i], dp_min[i-1] * nums[i], nums[i])
dp_min[i] = min(dp_max[i-1] * nums[i], dp_min[i-1] * nums[i], nums[i])
```

由此可得代码：

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        if len(nums) == 1:
            return nums[0]

        dp_max = [0] * len(nums)
        dp_min = [0] * len(nums)
        dp_max[0] = nums[0]
        dp_min[0] = nums[0]
        for i in range(1, len(nums)):
            dp_max[i] = max(
                dp_max[i-1] * nums[i],
                dp_min[i-1] * nums[i],
                nums[i]
            )
            dp_min[i] = min(
                dp_max[i-1] * nums[i],
                dp_min[i-1] * nums[i],
                nums[i]
            )
        return max(dp_max)
```

题型：买卖股票的最佳时机问题

> 从IV开始，需要通过定义每天的状态来进行dp推导。难点在于区分每天可能存在的不同状态

Eg1. 买卖股票的最佳时机

给定一个数组 `prices` ，它的第 `i` 个元素 `prices[i]` 表示一支给定股票第 `i` 天的价格。

你只能选择 **某一天** 买入这只股票，并选择在 **未来的某一个不同的日子** 卖出该股票。设计一个算法来计算你所能获取的最大利润。

返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 `0` 。

解：

从头往后遍历价格数组，并随时记录当前遇到的最低价格`min_price`和当前能获取的最高利润`max_profit`。每遍历到一个价格，首先更新当前的最低价格，然后再更新当前能拿到的最大利润：如果在当前天就卖出的话，那么当前能获得的最大利润一定是当天价格减去迄今为止的最低价格，和之前见过的最大利润做个比较即可。

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        max_profit = 0
        min_price = 10**6

        for i in range(len(prices)):
            min_price = min(min_price, prices[i])
            max_profit = max(max_profit, prices[i]-min_price)
        return max_profit
```

Eg1.2 买卖股票的最佳时机II

给你一个整数数组 `prices` ，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。你也可以先购买，然后在 **同一天** 出售。

返回 *你能获得的* ***最大*** *利润* 。

示例：

```
输入：prices = [7,1,5,3,6,4]
输出：7
解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5 - 1 = 4。
随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6 - 3 = 3。
最大总利润为 4 + 3 = 7 。
```

解：

只需确保所有“上坡”的利润都获取了即可：从头到尾遍历价格，每当当天价格大于前一天价格时，则将当天价格减去前一天价格的差作为利润加到总利润中

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if len(prices) == 0: return 0

        max_profit_all = 0
        for i in range(1, len(prices)):
            if prices[i] > prices[i-1]:
                max_profit_all += (prices[i] - prices[i-1])
        return max_profit_all
```

Eg1.3 买卖股票的最佳时机IV

给你一个整数数组 `prices` 和一个整数 `k` ，其中 `prices[i]` 是某支给定的股票在第 `i` 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 `k` 笔交易。也就是说，你最多可以买 `k` 次，卖 `k` 次。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例：

```
输入：k = 2, prices = [3,2,6,5,0,3]
输出：7
解释：在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出，这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
```

解：

在如此复杂的问题下，需要考虑如何设计每天的状态。每天的状态需要包括信息：到今天为止已经完成了多少笔交易，今天是否持有股票。

设`dp[i][j][0]`表示处于第`i`天且此时已完成`j`笔交易，且当天不持有股票时，能获得的最大利润；`dp[i][j][1]`为当天持有股票时能获得的最大利润。规定：1）卖出时才算是完成了1次交易，卖出当天的交易次数即变为j+1（设昨天为j），且当天的状态为不持有（0）；2）买入时不会影响交易次数j，只会使得当天持有股票的状态变成1（设昨天为0）。

则状态转移：

- 若`j==0`且当天未持有股票（`dp[i][j][0]`），也即之前未进行任何交易且今天也没买股票，则`dp[i][j][0]`只能和昨天保持一致：`dp[i][j][0] = dp[i-1][j][0]`

- 若`j>0`且当天未持有股票（`dp[i][j][0]`），则当天这种状态有可能由两种昨天的状态转换而来：1）若昨天也没持有股票，则说明今天没购买股票，和昨天（`dp[i-1][j][0]`）保持一致；2）若昨天持有股票，但今天卖出了导致今天变成了未持有股票的状态，则今天的`j`比昨天+1，也即昨天的状态（`dp[i-1][j-1][1]`）加上今天卖股票得到的钱（`prices[i]`）转移得到了今天的状态。转移公式：

```
dp[i][j][0] = max(
    dp[i-1][j][0],   # 前一天也没持有股票，则维持不变
    dp[i-1][j-1][1] + prices[i], # 前一天持有股票，但今天卖出了导致今天没持有股票，则挣回来今天卖股票的钱
)
```

- 若当天持有股票（`dp[i][j][1]`），则当天这种状态可能由两种昨天的状态转换而来：1）若昨天也持有股票，则说明今天没卖出股票（也即没完成交易），和昨天（`dp[i-1][j][1]`）保持一致；2）若昨天没持有股票，但今天购买了股票导致今天变成了持有股票的状态（注意购买股票并不算完成了一笔交易，因此昨天和今天的`j`不变），则昨天的状态（`dp[i-1][j][0]`）减去今天买股票花费的钱（`prices[i]`）转移得到了今天的状态。转移公式：

```
dp[i][j][1] = max(
    dp[i-1][j][1],  # 前一天也持有股票，则维持不变
    dp[i-1][j][0] - prices[i], # 前一天没持有股票，但今天买了股票导致今天持有了股票，则花出去今天买股票的钱
)
```

当到达最后一天时（此时一定已经卖出了股票，也即状态为`dp[n-1][j][0]`，因为最后一天还持有股票就纯属浪费了，把它卖了无论如何也能回点本），只需查看`0~k`这些交易次数中，哪种交易次数对应的最终利润最大即可。

另外，还可以进行的一种优化在于：`k>n//2`时，即使每天都进行买/卖交易也不会超出限额，则此时转化为k不受限制的问题（也即买卖股票的最佳时机II）

```
class Solution:
    def maxProfit(self, k: int, prices: List[int]) -> int:
        n = len(prices)
        if n == 1: return 0

        # 若 k>= n // 2，也即即使每天都进行买/卖交易也不会超出限额，则转化为k不受限制的问题（也即买卖股票的最佳时机II）
        if k >= n // 2:
            max_profit_all = 0
            for i in range(1, n):
                if prices[i] > prices[i-1]:
                    max_profit_all += prices[i] - prices[i-1]
            return max_profit_all

        # dp[i][j][0]表示处于第i天且此时已完成j笔交易时，此时不持有股票时的最大利润
        # dp[i][j][1]表示处于第i天且此时已完成j笔交易时，此时持有股票时的最大利润
        # 注：卖出时才算是完成了1次交易，卖出当天的交易次数即变为j+1（昨天为j），当天的状态为不持有股票(0)
        # 注：买入时不会影响交易次数j，只会使当天持有股票的状态变成1（昨天为0）
        # 在第i天买入一笔股票会使得总利润-prices[i]，卖出一笔股票会使得总利润+prices[i]
        dp = [[[-float('inf'), -float('inf')] for j in range(k+1)] for i in range(n)]
```

```
        # 初始化：dp[0][0][0] = 0 第0天已进行0次交易，且此时未持有股票
        # dp[0][0][1] = -prices[0] 第0天已进行0次交易，且当天买入了股票，此时持有股票
        dp[0][0][0] = 0
        dp[0][0][1] = -prices[0]

        for i in range(1, n):
            for j in range(k+1):
                # 若此时交易次数仍为0，且当天没买入股票（此时不持有股票），则前一天也必然没持有股票，和前一天保持不变
                if j == 0:
                    dp[i][j][0] = dp[i-1][j][0]
                else:
                    # 此时不持有股票的情况可能由如下两种状态转移而来
                    dp[i][j][0] = max(
                        dp[i-1][j][0],   # 前一天也没持有股票，则维持不变
                        dp[i-1][j-1][1] + prices[i], # 前一天持有股票，但今天卖出了导致今天没持有股票，则挣回来今天卖股票的钱
                    )
                # 此时持有股票的情况可能由如下两种状态转移而来
                dp[i][j][1] = max(
                    dp[i-1][j][1],  # 前一天也持有股票，则维持不变
                    dp[i-1][j][0] - prices[i], # 前一天没持有股票，但今天买了股票导致今天持有了股票，则花出去今天买股票的钱
                )

        # 遍历所有可取的k值（交易次数），选出在最后一天不持有股票的情况下最大的情况，即为总的最高利润
        max_profit_all = -float('inf')
        for j in range(k+1):
            max_profit_all = max(max_profit_all, dp[n-1][j][0])
        return max_profit_all
```

Eg1.4 买卖股票的最佳时机V

给你一个整数数组 `prices`，其中 `prices[i]` 是第 `i` 天股票的价格（美元），以及一个整数 `k`。

你最多可以进行 `k` 笔交易，每笔交易可以是以下任一类型：

- 普通交易：在第 `i` 天买入，然后在之后的第 `j` 天卖出，其中 `i < j`。你的利润是 `prices[j] - prices[i]`。
- 做空交易：在第 `i` 天卖出，然后在之后的第 `j` 天买回，其中 `i < j`。你的利润是 `prices[i] - prices[j]`。

注意：你必须在开始下一笔交易之前完成当前交易。此外，你不能在已经进行买入或卖出操作的同一天再次进行买入或卖出操作。

通过进行 最多 `k` 笔交易，返回你可以获得的最大总利润。

示例:

```
输入：prices = [1,7,9,8,2], k = 2
输出：14
解释：
我们可以通过 2 笔交易获得 14 美元的利润：
- 一笔普通交易：第 0 天以 1 美元买入，第 2 天以 9 美元卖出。
- 一笔做空交易：第 3 天以 8 美元卖出，第 4 天以 2 美元买回。
```

解：

本题相比于IV来说，增加了一种新的状态“做空状态”，也即在本来不持有股票的情况下卖出股票并获得当天股票的钱，然后处于“持有股票数=-1”的做空状态，等后边将其买回并损失买股票所花的钱后，恢复普通的不持有股票状态。由此只需添加一种`dp[i][j][2]`状态来表示做空状态，推导关系详见注释

```python
class Solution:
    def maximumProfit(self, prices: List[int], k: int) -> int:
        n = len(prices)
        if n == 1: return 0

        # dp[i][j][0]表示处于第i天且此时已完成j笔交易时，此时不持有股票时的最大利润
        # dp[i][j][1]表示处于第i天且此时已完成j笔交易时，此时持有股票时的最大利润
        # dp[i][j][2]表示处于第i天且此时已经完成j笔交易时，此时处于做空状态的最大利润
        # 在第i天买入一笔股票会使得总利润-prices[i]，卖出一笔股票会使得总利润+prices[i]
        dp = [[[-float('inf'), -float('inf'), -float('inf')] for j in range(k+1)] for i in
range(n)]
        # 初始化：dp[0][0][0] = 0 第0天已进行0次交易，且此时未持有股票
        # dp[0][0][1] = -prices[0] 第0天已进行0次交易，且当天买入了股票，此时持有股票
        # dp[0][0][2] = prices[0] 第0天已进行了0次交易，且当天卖出了股票，此时处于做空状态
        dp[0][0][0] = 0
        dp[0][0][1] = -prices[0]
        dp[0][0][2] = prices[0]

        for i in range(1, n):
            for j in range(k+1):
                # 若此时交易次数仍为0，且当天没买入股票（此时不持有股票），则前一天也必然没持有股票，和
前一天保持不变
                if j == 0:
                    dp[i][j][0] = dp[i-1][j][0]
                else:
                    # 此时不持有股票的情况可能由如下3种状态转移而来
                    dp[i][j][0] = max(
                        dp[i-1][j][0],    # 前一天也没持有股票，则维持不变
                        dp[i-1][j-1][1] + prices[i], # 前一天持有股票，但今天卖出了导致今天没持有
股票，则挣回来今天卖股票的钱
                        dp[i-1][j-1][2] - prices[i]      # 前一天为做空状态，但今天买回了导致今天
恢复普通的没持有股票状态，则损失今天买回股票的钱
                    )
                # 此时持有股票的情况可能由如下两种状态转移而来
                dp[i][j][1] = max(
                    dp[i-1][j][1],  # 前一天也持有股票，则维持不变
                    dp[i-1][j][0] - prices[i], # 前一天没持有股票，但今天买了股票导致今天持有了股
票，则花出去今天买股票的钱
```

```
                )
                # 此时处于做空的情况，可能由以下两种状态转移而来
                dp[i][j][2] = max(
                    dp[i-1][j][0] + prices[i],   # 前一天没持有股票，今天卖出了股票，获得了卖出股票的钱
                    dp[i-1][j][2]   # 前一天也处于做空状态
                )

        # 遍历所有可取的k值（交易次数），选出在最后一天不持有股票的情况下最大的情况，即为总的最高利润
        max_profit_all = -float('inf')
        for j in range(k+1):
            max_profit_all = max(max_profit_all, dp[n-1][j][0])
        return max_profit_all
```

Eg1.5 买卖股票的最佳时机含冷冻期

给定一个整数数组`prices`，其中第 `prices[i]` 表示第 `i` 天的股票价格 。

设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：

卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。

注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例：

```
输入：prices = [1,2,3,0,2]
输出：3
解释：对应的交易状态为：[买入, 卖出, 冷冻期, 买入, 卖出]
```

解：

本题的状态也比较复杂，对于每天来说，其基本的两个状态肯定是今天持有股票/不持有股票。然而，由于冷冻期的限制，今天不持有股票的情况会在“今天卖出了股票导致今天没股票”的子情况下影响明天的买卖（这样的话明天就成了冷冻期），因此还要进一步分为两种子情况：1）今天不持有股票，且并不是因为今天卖出了股票，而是之前本来就没股票，则明天不是冷冻期；2）今天不持有股票，是因为今天卖出了股票，会导致明天是冷冻期。

因此，定义状态：`dp[i][0]`表示今天持有股票，`dp[i][1]`表示今天不持有股票（且并不是因为今天卖出了股票，也即明天非冷冻期），`dp[i][2]`表示今天不持有股票（是因为今天卖出了股票，也即明天是冷冻期）

状态转移：

- 今天持有股票（`dp[i][0]`）：其可能来源于两种昨天的情况：1）可能是因为昨天本来就有股票，今天没变，也即等于`dp[i-1][0]`；2）可能是昨天本来没股票，今天现买的（因此需要确保昨天是1状态而不是2状态），则今天需要花费买股票的钱，也即等于`dp[i-1][1]-prices[i]`
- 其他两种状态转移详见注释

```
```

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0

        n = len(prices)
        # dp[i][0]：第i天且当天持有股票（可能是今天买的，也可能是之前就有的）时的最大收益
        # dp[i][1]：第i天且当天不持有股票（并不是因为今天卖出股票，也即明天不是冷冻期）时的最大收益
        # dp[i][2]：第i天且当前不持有股票（是因为今天卖出了股票，也即明天是冷冻期）
        dp = [[-float('inf'), -float('inf'), -float('inf')] for _ in range(n)]
        # 初始化：
        # dp[0][0] = -prices[0]  第0天就买了股票花了钱，使得第1天就持有了股票
        # dp[0][1] = 0  第0天没买股票，也没花钱
        # dp[0][2] 不可能出现，因为第0天不可能卖出股票，仍为负无穷
        dp[0][0] = -prices[0]
        dp[0][1] = 0
        dp[0][2] = -float('inf')

        for i in range(1, n):
            # 如果今天持有股票，则可能是：1）昨天没股票，今天现买的（需要昨天是1状态，确保今天不是冷冻期），则今天需要花费买股票的钱；2）昨天本来就有股票，今天没变，则保持昨天有股票的状态
            dp[i][0] = max(dp[i-1][0], dp[i-1][1] - prices[i])
            # 如果今天不持有股票且今天没卖股票，则只能是昨天也没股票，今天没变，保持昨天没股票的状态（此时昨天是1还是2状态都无所谓，反正今天没卖股票），不可能是昨天有股票的情况（因为今天没卖股票，所以这种情况下今天不可能是未持有股票的状态）
            dp[i][1] = max(dp[i-1][1], dp[i-1][2])
            # 如果今天不持有股票且今天卖了股票，则昨天必须有股票（才能今天卖），也即必须从0状态转换而来，且今天获得了卖出股票的收入
            dp[i][2] = dp[i][0] + prices[i]

        # 最后一天一定是不持有股票的情况(1或2状态)，最后一天及之前一定已经把股票处理掉了
        max_profit_all = max(dp[n-1][1], dp[n-1][2])
        return max_profit_all
```

Eg1.6 买卖股票的最佳时机含手续费

给定一个整数数组 `prices`，其中 `prices[i]`表示第 `i` 天的股票价格 ； 整数 `fee` 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

解：

虽然交易次数不限，但由于进行交易会带来手续费，因此不能像II那样只考虑买卖股票的差价，简单地统计上坡数量，因此仿照前边那几道比较复杂的题，通过定义每天的状态来进行dp。同样，用`dp[i][0]`表示第i天不持有股票时的最大利润，`dp[i][1]`表示第i天持有股票时的最大利润，则转移关系跟前边的题也类似，只不过卖出股票时要加入手续费损失项

```python
class Solution:
```

```python
    def maxProfit(self, prices: List[int], fee: int) -> int:

        n = len(prices)
        # dp[i][0]: 第i天不持有股票时的最大利润，dp[i][1]：第i天持有股票时的最大利润
        dp = [[-float('inf'), -float('inf')] for _ in range(n)]
        dp[0][0] = 0
        dp[0][1] = -prices[0]

        for i in range(1, n):
            dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i] - fee)
            dp[i][1] = max(dp[i-1][0] - prices[i], dp[i-1][1])

        return dp[n-1][0]
```

题型：字符串变换匹配问题

Eg 1. 编辑距离

给你两个单词 `word1` 和 `word2`， *请返回将 `word1` 转换成 `word2` 所使用的最少操作数* 。

你可以对任意一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

解：

原始情况下，一共有6种操作：对word1进行插入/删除/替换、对word2进行插入/删除/替换，较为复杂。观察发现它们其实是存在冗余的：从最终结果来看，对word1的插入某个字符等价于对word2删除一个字符、对word1删除某个字符等价于对word2插入一个字符、对word1替换某个字符等价于对word2替换某个字符。

因此，可以将操作简化为3种：在word1中添加一个字符、在word2中添加一个字符、在word1中替换一个字符。

进而可以设置`dp[i][j]`为：`word1[:i]`与`word2[:j]`这两个子串（开区间，不包括`word1[i],word2[j]`）之间的最小距离。这样一来，`dp[i][j]`的来源可能有3种：

- `dp[i-1][j]+1`：也即`word1[:i-1]`与`word2[:j]`的最小距离，只需再多1步即可变为`dp[i][j]`（`word1[:i]`删除1个字符变成`word1[:i-1]`，然后经过`dp[i-1][j]`步到达`word2[:j]`）
- `dp[i][j-1]+1`：同理
- `dp[i-1][j-1]+1`或`dp[i-1][j-1]`：如果`word1[i]==word2[j]`的话，那么知道`dp[i-1][j-1]`后可以直接变到`dp[i][j]`（因为它们两个末尾的字符本来就相等，无需变化）；如果`word1[i]!=word2[j]`，则`dp[i-1][j-1]+1=dp[i][j]`（将末尾字符`word1[i]`替换成`word2[j]`，然后经过`dp[i-1][j-1]`步使得前边的`word1[:i-1]`和`word2[:j-1]`对齐）

边界条件：从一个空串到另一个字符串的距离就是另一个字符串的长度（在空串基础上依次添加另一个字符串的长度），因此边界条件即为所有`dp[i][0]`和`dp[0][k]`

```
```

```python
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        # 极端情况：其中一个为空串，则其到另一个串的最短路径就是依次添加另一个串的字符直到变成另一个串
        if len(word1) == 0 or len(word2) == 0:
            return len(word1) if len(word2) == 0 else len(word2)

        # dp[i][j]表示从word1[:i]到word2[:j]的最短距离（闭区间，包括右端点）
        # 注意由于是从[0][0]推到[len(word1)][len(word2)]，因此dp数组长度为(len(word1)+1,
len(word2)+1)
        dp = [[0 for _ in range(len(word2)+1)] for _ in range(len(word1)+1)]

        # 边界条件：从空串到另一个串的距离就是另一个串的长度
        for i in range(len(word1)+1):
            dp[i][0] = i
        for j in range(len(word2)+1):
            dp[0][j] = j

        for i in range(1, len(word1)+1):
            for j in range(1, len(word2)+1):
                # 考察word1[i-1]到word2[j-1]的距离，也即求dp[i][j]
                if word1[i-1] == word2[j-1]:
                    # 如果两段字串最后一个字符是一样的，则可直接从dp[i-1][j-1]变过来
                    dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1])
                else:
                    dp[i][j] = min(dp[i-1][j]+1, dp[i][j-1]+1, dp[i-1][j-1]+1)

        return dp[len(word1)][len(word2)]
```

Eg2. 正则表达式匹配

给你一个字符串 `s` 和一个字符规律 `p`，请你来实现一个支持 `'.'` 和 `'*'` 的正则表达式匹配。

- `'.'` 匹配任意单个字符
- `'*'` 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖 **整个** 字符串 `s` 的，而不是部分字符串。

```
示例 1：
输入：s = "aa", p = "a"
输出：false
解释："a" 无法匹配 "aa" 整个字符串。

示例 2：
输入：s = "aa", p = "a*"
输出：true
```

解释：因为 '*' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。

示例 3：
输入：s = "ab", p = ".*"
输出：true
解释：".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。

解：

设`dp[i][j]`表示`s`当前缀部分`s[:i]`和模式串`p`当前缀部分`p[:j]`（不包括`s[i],p[j]`）是否能匹配。则分别初始化`s`,`p`分别为空串的情况（`dp[i][0], dp[0][j]`）后，逐次推导各个`dp[i][j]`。核心在于区分新加入模式串的`p[j-1]`是否为重复符`'*'`的情况，然后再继续考虑其子情况。转移关系详见注释：

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)
        # dp[i][j]表示s[:i]和p[:j]（左闭右开）能否匹配
        dp = [[False for i in range(n+1)] for j in range(m+1)]
        # 初始化，空串和空串肯定能匹配
        dp[0][0] = True
        # 初始化首行，也即s为空串时，查看p的各个前缀子串p[:j]能否和它匹配
        # 由于s是空串，因此仅当j为偶数且p[:j]这段中的所有偶数位都是*时（使得对应的每个奇数位字符都出现0次），才能匹配空串。j为奇数时肯定匹配不了，dp[0][j]必然为False直接跳过即可
        for j in range(2, n+1, 2):    # 遍历所有的偶数j
            if p[j - 1] != '*':
                break   # 如果发现了一个p[j-1]不是'*'，则从此往后的dp[0][j]就都是False了
            else:
                dp[0][j] = True

        # 初始化首列，也即模式串p为空时。此时如果s不是空串的话，那必然匹配不上。因此除了dp[0][0]外的所有dp[i][0]都是False。
        # 由于初始化dp时就全都赋的False，因此这里不再需要重新初始化一遍。

        # 然后从i=1, j=1开始遍历各个s[:i], p[:j]
        for i in range(1, m+1):
            for j in range(1, n+1):
                # 以加入模式串p[:j-1]的新字符p[j-1]是否等于'*'进行大的情况分类
                # 1.模式串末尾p[j-1] == '*'时的情况
                if p[j-1] == '*':
                    # 首先考虑s[:i]不动，模式串缩短为p[:j-2]时，如果dp[i][j-2]==True，也即s[:i]和p[:j-2]可以匹配，那么p[j-1]='*'的情况下只需让p[j-2]出现0遍即可，也即'*'的作用是x0，这样就相当于s[:i]和p[:j-2]匹配，因此dp[i][j]也是True
                    if dp[i][j-2] == True:
                        dp[i][j] = True
                    # 然后考虑模式串p[:j]不动，s缩短为s[:i-1]的情况，如果s[:i-1]可以匹配p[:j],然后在s[:i-1]上加入s[i-1]后:
                    elif dp[i-1][j] == True:
                        # 如果s[i-1]恰好就等于p[j-2]，由于p[j-1]='*'，因此只需让p[j-2]再多出现一次即可匹配掉s[i-1]
```

```
                        # 或者如果p[j-2]为万能的'.'，那么也相当于等于s[i-1]，同样利用p[j-1]='*'使得p[j-2]多出现一次以匹配掉s[i-1]
                        if s[i-1] == p[j-2] or p[j-2] == '.':
                            dp[i][j] = True

                # 2.考虑p[j-1] != '*'时的情况
                else:
                    # 如果dp[i-1][j-1]==True，也即s[:i-1]和p[:j-1]能匹配，那么当新加入模式串的p[j-1]恰好和新加入s的s[i-1]相同时（或p[j-1]为万能的'.'时），s[:i]和p[:j]仍然可以匹配上
                    if dp[i-1][j-1] == True:
                        if s[i-1] == p[j-1] or p[j-1] == '.':
                            dp[i][j] = True
        return dp[-1][-1]
```

其他dp问题

各种奇怪的状态转移

Eg2：打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，**如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警**。

给定一个代表每个房屋存放金额的非负整数数组，计算你 **不触动警报装置的情况下** ，一夜之内能够偷窃到的最高金额。

示例：

```
输入：[1,2,3,1]
输出：4
解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
     偷窃到的最高金额 = 1 + 3 = 4 。
```

解：

设`dp[i]`表示总共有i家时能偷到的最大金额

当面对第`i`家时：

- 若选择偷窃第`i`家，则不能偷窃第`i-1`家，则此时最大总金额为：`dp[i-2] + nums[i]`
- 若选择不偷窃第`i`家，则此时最大金额就是偷窃前`i-1`家的最大金额，也即：`dp[i-1]`

初始条件：`dp[0]=nums[0]`，`dp[1]=max(nums[0], nums[1])`

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        if len(nums) == 1: return nums[0]
        if len(nums) == 2: return max(nums)

        dp = [0] * len(nums)
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])
        for i in range(2, len(nums)):
            dp[i] = max(dp[i-2]+nums[i], dp[i-1])

        return dp[len(nums)-1]
```

Eg2.2 打家劫舍II

改成“这个地方所有的房屋都 **围成一圈** ”，也即房屋链首尾相连

解：

由于首尾相连，因此不能同时偷`nums[0]`和`nums[-1]`，如果偷`nums[0]`的话相当于可偷范围缩小到了`nums[0]~nums[-2]`，如果偷`nums[1]`的话相当于可偷范围缩小到了`nums[1]~nums[-1]`。因此，只需构造两个dp数组，分别对上述两种情况做dp，最终再取二者中较大的结果即可

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        if len(nums) == 1: return nums[0]
        if len(nums) == 2: return max(nums)

        # nums[0] ~ nums[-2]
        dp_1 = [0] * (len(nums) - 1)
        dp_1[0] = nums[0]
        dp_1[1] = max(nums[0], nums[1])
        for i in range(2, len(nums)-1):
            dp_1[i] = max(dp_1[i-1], dp_1[i-2]+nums[i])

        # nums[1] ~ nums[-1]
        dp_2 = [0] * (len(nums) - 1)
        dp_2[0] = nums[1]
        dp_2[1] = max(nums[1], nums[2])
        for i in range(2, len(nums)-1):
            dp_2[i] = max(dp_2[i-1], dp_2[i-2]+nums[i+1])

        return max(dp_1[-1], dp_2[-1])
```

双指针

Eg1: 合并两个有序数组

给你两个按 **非递减顺序** 排列的整数数组 `nums1` 和 `nums2`，另有两个整数 `m` 和 `n`，分别表示 `nums1` 和 `nums2` 中的元素数目。

请你 **合并** `nums2` 到 `nums1` 中，使合并后的数组同样按 **非递减顺序** 排列。

注意：最终，合并后数组不应由函数返回，而是存储在数组 `nums1` 中。为了应对这种情况，`nums1` 的初始长度为 `m + n`，其中前 `m` 个元素表示应合并的元素，后 `n` 个元素为 `0`，应忽略。`nums2` 的长度为 `n`。

解：

使用逆向双指针：`p1`,`p2`初始时分别指向`nums1`,`nums2`数据部分末尾（也即`m-1`,`n-1`部分），然后使用指针`tail`指向`nums1`的最末尾（`m+n-1`）。然后将`p1,p2`依次往前移动：每次都比较一下`nums1[p1]`和`nums2[p2]`哪个大，选取大的填入`nums1[tail]`，然后将相应的`p1/p2`和`tail`均向前移动一位

```
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        if len(nums2) == 0: return
        if len(nums1) == 0:
            nums1 = nums2
            return

        tail, p1, p2 = m + n - 1, m - 1, n - 1
        while True:
            if nums1[p1] >= nums2[p2]:
                nums1[tail] = nums1[p1]
                p1 -= 1
                if p1 < 0:
                    nums1[:p2+1] = nums2[:p2+1]
                    return
            elif nums1[p1] < nums2[p2]:
                nums1[tail] = nums2[p2]
                p2 -= 1
                if p2 < 0:
                    return
            tail -= 1
        return
```

Eg2. 移动零

给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意，必须在不复制数组的情况下原地对数组进行操作。

示例:

```
输入: nums = [0,1,0,3,12]
输出: [1,3,12,0,0]
```

解：

本题同样需要原地更改数组，因此类似上一题，也可以设置两个指针：一个指针负责逐个遍历原数组元素，另一个指针负责指向下一个非0元素在最终排好的数组中应该处于的位置（也即该非0元素在原数组所有非0元素中所处的相对位置）。其中后者本质上是一个计数器，其统计当前已经遍历到的非0元素的数量，则这个数量值就是下一个要遍历到的非0元素在所有非0元素中的相对位置，也就是其在排好数组中的绝对索引。

因此，首先设置非0元素数量计数器`non_zero_num=0`，然后开始从头到尾遍历原数组：若当前元素`nums[i]`非0，则将其插入到`nums[non_zero_num]`位置，然后将计数器`non_zero_num += 1`。这里不需要担心插入过程会覆盖掉`nums[non_zero_num]`原先的元素，因为任何时刻只有`nums[0:non_zero_num]`中的元素是放好的非0元素，而`nums[non_zero_num: i]`这一段一定都是剩下的0，因此可以直接覆盖掉。

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        # 一个指针负责指向下一个非0元素在排好的数组中应该处于的绝对位置
        # 也即：统计当前已遍历到多少个非0元素了，它代表遍历到的下一个非0元素在所有非0元素中的相对位置，也即下一个非0元素在最终排好的数组中的绝对位置
        non_zero_num = 0

        # 一个指针负责遍历原数组nums中的各个元素
        for i in range(len(nums)):
            if nums[i] != 0:    # 如果当前元素非0，则将其插入到其在最终排好的数组中的绝对位置，也就是指针non_zero_num指向的位置
                nums[non_zero_num] = nums[i]
                non_zero_num += 1   # 然后更新non_zero_num指向的位置（或者说当前遍历到的非0元素的数量）
        # 遍历完成之后，此时nums[0: non_zero_num]中已经按照顺序填充好了所有非0元素，只需再把剩余位置nums[non_zero_num: ]填充为0即可
        nums[non_zero_num: ] = [0] * (len(nums) - non_zero_num)
```

Eg2. 删除排序数组中的重复项

给你一个 **非严格递增排列** 的数组 `nums` ，请你原地删除重复出现的元素，使每个元素 **只出现一次** ，返回删除后数组的新长度。元素的 **相对顺序** 应该保持 **一致** 。然后返回 `nums` 中唯一元素的个数。

考虑 `nums` 的唯一元素的数量为 `k`。去重后，返回唯一元素的数量 `k`。

`nums` 的前 `k` 个元素应包含 **排序后** 的唯一数字。下标 `k - 1` 之后的剩余元素可以忽略。

示例：

```
输入：nums = [0,0,1,1,1,2,2,3,3,4]
输出：5, nums = [0,1,2,3,4,_,_,_,_,_]
解释：函数应该返回新的长度 5 ， 并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4 。不需要考虑数组中超出新长度后面的元素。
```

解：

本题也是原地移动问题，思路和"移动零"相同，维护一个慢指针（可作为计数器，其之前都是已经放好的非重复元素）`slow`和一个快指针`fast`，不断移动`fast`，如果发现当前元素和它前边的元素不相等，则说明发现了一个新的不重复元素，因此将其赋给`slow`位置进行记录，然后令计数器`slow+1`；若当前元素是重复元素则只给`fast+1`，来继续查看下一个元素

```
class Solution:
    def removeDuplicates(self, nums: List[int]) -> int:
        if not nums:
            return 0

        slow = 1    # slow及之前的元素都是唯一元素
        fast = 1    # fast为当前遍历到的元素位置
        while fast < len(nums):
            if nums[fast] != nums[fast - 1]:    # 如果nums[fast]和前一个元素不重复，那么就将其记录到slow当前的位置，并把slow+1（也即已知的不重复元素数量+1）
                nums[slow] = nums[fast]
                slow += 1
            fast += 1   # fast+1来查看下一个元素
        # 最终slow的值就是统计到的不重复元素数量
        return slow
```

Eg4. 颜色分类

给定一个包含红色、白色和蓝色、共 `n` 个元素的数组 `nums` ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 `0` 、 `1` 和 `2` 分别表示红色、白色和蓝色。

必须在不使用库内置的 sort 函数的情况下解决这个问题。

示例：

```
输入：nums = [2,0,2,1,1,0]
输出：[0,0,1,1,2,2]
```

解：

本题和"移动零"类似，都是需要原地对数组进行重排。由于本题涉及了3种元素（最终形成3个区域），因此使用3个指针：`left, cur, right`，初始化为`0, 0, len(nums)-1`，由`cur`表示当前遍历到的位置，发现0则将其往左扔，发现2则将其往右扔，发现1则将其保留原地，而`left,right`则界定了当前数组左端累积的0的范围和数组右端累积的2的范围。

具体而言，`left`左边全为0，其从左往右移动；`cur`也从左往右移动，其指向的当前位置如果是0，则和`nums[left]`交换，如果是2则和`nums[right]`交换，如果是1则不需要交换，将这个1留在原地即可；`right`右边全是2，其从右往左移动。

这样一来，这会使得`left~cur`之间这一段全是1，`cur~right`之间这一段是待处理的剩余部分，当`cur`和`right`相遇时说明所有剩余部分都处理干净了。

需要注意往左扔0或往右扔2后，在`left+=1`或`right-=1`之余，是否需要将`cur`右移一位：如果是往左扔0的话则需要同时将`cur+=1`，因为`nums[left]`早已被`cur`经过过了，它肯定是1，因此将其换到`nums[cur]`后就留在原地就行了；但如果是往右扔2的话，由于并不知道`nums[right]`是什么数字，因此将其换到`nums[cur]`后不能不管，需要在下一轮循环中处理它，因此这一步不需要将`cur+=1`

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        # left左边全为0，其从左往右移动
        # cur也从左往右移动，其指向的当前位置如果是0，则和nums[left]交换，如果是2则和nums[right]交换，如果是1则不需要交换，将这个1留在原地即可
        # right右边全是2，其从右往左移动
        # 这会使得left~cur之间这一段全是1，cur~right之间这一段是待处理的剩余部分，当cur和right相遇时说明所有剩余部分都处理干净了
        left, cur, right = 0, 0, len(nums) - 1

        # 不断把0往左扔，把2往右扔，直到从左到右的cur和从右往左的right相遇，表示剩余部分都处理清了
        while cur <= right:
            if nums[cur] == 0:
                # 若nums[cur]==0，则将其和nums[left]交换
                # 由于left和cur都是从左往右移动，因此nums[left]位置一定早已被nums[cur]经过过了，所以如果cur<left的话，它一定是1，因此换过来后就不用再管这个新的nums[cur]（原先的nums[left]）了，可以同时把cur和left右移一位
                nums[cur], nums[left] = nums[left], nums[cur]
                cur += 1
                left += 1
            elif nums[cur] == 2:
                # 若nums[cur]==2，则将其和nums[right]交换
                # 注意，由于right是从右端往左移动的，因此nums[right]是什么并不知道，它可能是0或2，因此把它和nums[cur]调换后，新的nums[cur]（原先的nums[right]）不能被跳过，还需要在下一轮循环中看一下它是什么数字并决定要不要继续调换，因此这一步交换完后不能动cur，只是把right左移一位
                nums[cur], nums[right] = nums[right], nums[cur]
                right -= 1
            elif nums[cur] == 1:
                cur += 1
```

Eg5. 移除元素

给你一个数组 `nums` 和一个值 `val`，你需要 原地 移除所有数值等于 `val` 的元素。元素的顺序可能发生改变。然后返回 `nums` 中与 `val` 不同的元素的数量。

假设 `nums` 中不等于 `val` 的元素数量为 `k`，要通过此题，您需要执行以下操作：

- 更改 `nums` 数组，使 `nums` 的前 `k` 个元素包含不等于 `val` 的元素。`nums` 的其余元素和 `nums` 的大小并不重要。
- 返回 `k`。

示例：

```
输入：nums = [0,1,2,2,3,0,4,2], val = 2
输出：5, nums = [0,1,4,0,3,_,_,_]
```

解：

本题和前几题思路一样，分别维护一个左指针和一个右指针，其中左指针左侧为当前已经发现且移到最前边的那些非`val`的元素，其指向位置为下一个非`val`元素应该被复制到的位置，右指针为当前探索的前沿。每次都查看右指针指向元素是否等于`val`，若等于则仅右指针右移一位，左指针不动；若不等于则将该元素复制到左指针指向的位置，然后左右指针都右移一位：

```python
class Solution:
    def removeElement(self, nums: List[int], val: int) -> int:

        left, right = 0, 0

        while right < len(nums):
            if nums[right] == val:
                # 若当前处理的元素等于val，则仅右指针继续右移一位，左指针不动
                right += 1
            else:
                # 若当前处理的元素不等于val，则将其复制到当前左指针指向的位置，然后左右指针都右移一位
                nums[left] = nums[right]
                left += 1
                right += 1

        return left
```

Eg6. 最短无序连续子数组

给你一个整数数组 `nums` ，你需要找出一个 **连续子数组** ，如果对这个子数组进行升序排序，那么整个数组都会变为升序排序。

请你找出符合题意的 **最短** 子数组，并输出它的长度。

示例：

```
输入：nums = [2,6,4,8,10,9,15]
输出：5
解释：你只需要对 [6, 4, 8, 10, 9] 进行升序排序，那么整个表都会变为升序排序。
```

解：

设置一个左指针`left`和一个右指针`right`，分别从数组两端开始往中间移动，并确保：`left`及其左侧区间是升序的，`right`及其右侧区间也是升序的，而`left~right`之间则是无序区间，同时保证左侧区间的最大值小于等于无序区间中的最小值，右侧区间的最小值大于等于无序区间中的最大值，此时的无序区间排成升序后就能确保整个数组都是升序的了。

因此，首先将`left`从左往右移动，并找到第一个破坏升序的位置，然后将`right`从右往左移动，并找到第一个破坏（从右往左）降序的位置。

然而，此时找到的中间无序区间并不一定能符合“区间中最小值大于等于左侧区间最大值、区间中最大值小于等于右侧区间最小值”，例如：`[1,3,4,[6,2,10,7],8,9]`将中间区间排好后整个数组就成了`[1,3,4,[2,6,7,10],8,9]`，由于区间中最小值`2`比左侧区间最大值`4`小、最大值`10`比右侧区间最小值`10`大，导致此时整个数组中并不能连续升序，只是三个区间各自升序，再区间交界处出现了失序情况。因此，在上一步得到的中间区间的基础上，求得中间区间的最小值和最大值后，进一步将`left`向左和`right`向右移动一定距离，使得中间区间进行扩张，直到确保左侧区间的所有值都小于等于中间区间最小值、右侧区间所有值都大于等于中间区间最大值。

```python
class Solution:
    def findUnsortedSubarray(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1

        # 首先找到初步的无序区间，左右端点分别为左侧和右侧第一次破坏升序关系的位置
        while left < len(nums) - 1:
            if nums[left] <= nums[left + 1]:
                left += 1
            else:
                break

        # 如果整个序列本身就是升序
        if left == len(nums) - 1:
            return 0

        while right > 0:
            if nums[right] >= nums[right - 1]:
                right -= 1
            else:
                break

        # 此时无序区间为nuns[left]~nums[right]（左闭右闭）
        # 算出当前无序区间的最小值和最大值，并分别扩展左右边界，直到确保左区间最大元素小于等于区间最小值，右区间最小元素大于等于区间最大值
        min_val = min(nums[left: right+1])
        max_val = max(nums[left: right+1])

        while left > 0:
            if nums[left-1] > min_val:
                left -= 1
            else:
                break
        while right < len(nums) - 1:
            if nums[right+1] < max_val:
                right += 1
            else:
                break

        return right - left + 1
```

Eg7：三数之和

给你一个整数数组 `nums` ，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k` ，同时还满足 `nums[i] + nums[j] + nums[k] == 0` 。请你返回所有和为 `0` 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

例 1：

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
解释：
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。
不同的三元组是 [-1,0,1] 和 [-1,-1,2]
```

解：不让输出重复的三元组会比较麻烦。例如：`[1,2,3,4,2,-1,-1]`，若直接使用三重循环的话，则由于first会遍历到两次2，所以会得到两个`[2,-1,-1]`。因此可以先将数组进行排序，这样比较容易判重。

具体而言：对于排序后的数组，首先遍历第1个数字（first），如果发现本轮的first和上一轮的first相同则说明重复，可以跳过本轮遍历。

在first的循环节内部，进一步设置second（初始为`first+1`）和third（初始为`len(nums)-1`）两个指针，它们分别从左到右和从右到左移动。由于此时的序列有序，因此很容易判重以及在合适时机跳出循环

```python
class Solution(object):
    def threeSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        # 先排序，防止重复
        nums.sort()
        n = len(nums)
        res = []

        # 枚举 first
        for first in range(n):
            # 去重，如果本轮的first和上一轮的first一样，那么即使后边的second，third有满足要求的，也是重复结果，因此跳过。
            if first > 0 and nums[first] == nums[first-1]:
                continue
            third = n - 1
            target = -nums[first]
            # 枚举second（或者说一位一位向右移动second）
            for second in range(first+1, n):
                # 去重，如果本轮的second和上一轮的second一样，那么即使后边的third有满足要求的，也是重复结果，因此跳过
```

```
                if second > first + 1 and nums[second] == nums[second-1]:
                    continue
                # 开始从右向左移动third指针，同时保证third一直在second右侧。当nums[second] + nums[third] > target时不停向左移动third
                while second < third and nums[second] + nums[third] > target:
                    third -= 1
                # 若是因为second == third重合而退出的循环，则说明这个first下在此second往后都不可能有nums[second] + nums[third] <= target的情况了（因为序列是递增的），因此可以跳过该first后续的second的遍历了
                if second == third:
                    break
                # 若是因为找到一组加起来等于target的second和third而退出循环的，则将其记录
                if nums[second] + nums[third] == target:
                    res.append([nums[first], nums[second], nums[third]])
        return res
```

Eg8. 盛最多水的容器

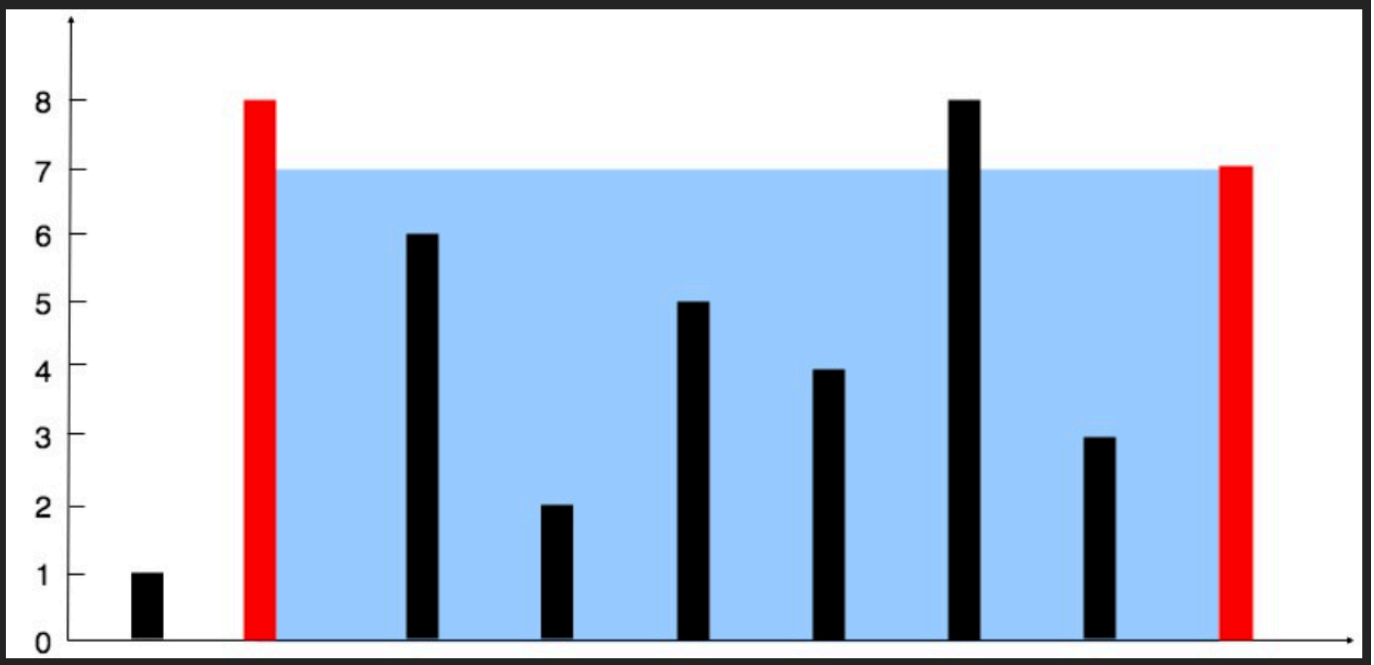
给定一个长度为 `n` 的整数数组 `height` 。有 `n` 条垂线，第 `i` 条线的两个端点是 `(i, 0)` 和 `(i, height[i])` 。

找出其中的两条线，使得它们与 `x` 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1：



> **输入：** [1,8,6,2,5,4,8,3,7]
> **输出：** 49
> **解释：** 图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

解：

该问题中，由于水可以淹没中间的柱子，因此和接雨水的解决思路完全不同。对于一个容器区间来说，影响该区间盛水量的因素有：1）左右两边柱子中较矮一边的高度；2）两个柱子之间的距离。

因此，一开始将左右指针分别放在最左边和最右边，然后开始逐步往中间缩小容器宽度：每次统计完当前容器盛水量后，就将左右指针中对应高度较矮的一方往中心移动1位。这是因为：如果移动较高的一边的话，由于较矮的一边没变化，因此移动后不管高的一边如何变化，新容器的容积也不可能增大（相当于继承了原容器短板的同时还缩小了宽度）。因此，只有移动矮的一方的指针，才有可能将这一边的高度拉高，从而在缩小宽度后还有容积增大的可能性。

```
class Solution:
    def maxArea(self, height: List[int]) -> int:
        max_area = 0
        left, right = 0, len(height) - 1

        # 初始时左右指针分别在最左侧和最右侧
        while left < right:
            area = (right - left) * min(height[right], height[left])
            max_area = max(max_area, area)
            if height[left] > height[right]:    # 如果左端高于右端，则将右端（矮的一边）往左移动1
位
                right -= 1
            else:   # 否则，将左端往右移动1位
                left += 1
        return max_area
```

滑动窗口

Eg1：无重复字符的最长子串

```
class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        if len(s) == 0:
            return 0
        if len(s) == 1:
            return 1

        max_len = 1
        seen = set()     # 维护当前滑动窗口中见过的字符
        right_prev = 1  # 如果上一轮（左端点的循环）是因为右端字符在seen中重复而退出的，则记录这个退出点，下次继续从这个点开始向右扩张（避免每次都从left+1开始向右推）

        for left in range(len(s)-1):     # 逐次移动窗口的左端点
            seen.add(s[left])   # 将左端点加入seen
            break_flag = False
            for right in range(max(left+1, right_prev), len(s)): # 移动右端点
                if s[right] in seen: # 如果右端点重复了，则退出
                    break_flag = True
                    break
                else:   # 如果右端点没重复，则将其加入seen并继续往右扩张
                    seen.add(s[right])
            if break_flag: # 本轮对于右端点的扩张结束后，如果是因为右端点重复而结束的，则将right_prev设为right，更新最大长度，然后继续移动左端点（并将本次的左端点在seen中删除）
                max_len = max(max_len, right - left)
                right_prev = right
                seen.remove(s[left])
            else:  # 如果本轮右端点到达了字符串尽头，则在本次的左端点之右的左端点不可能带来更长的无重复子串，因此更新最大长度并退出即可
                max_len = max(max_len, right - left + 1)
                break
        return max_len
```

Eg2：长度最小的子数组：

给定一个含有 `n` 个正整数的数组和一个正整数 `target` 。

找出该数组中满足其总和大于等于 `target` 的长度最小的子数组`[numsl, numsl+1, ..., numsr-1, numsr]`，并返回其长度。如果不存在符合条件的子数组，返回 `0` 。

解：

```
class Solution:
    def minSubArrayLen(self, target: int, nums: List[int]) -> int:
        if len(nums) == 1: return 1 if nums[0] >= target else 0
        min_len = 10**6

        left = 0
        right = 0
        total = 0
        # 右指针不断挣钱，挣到当前数组>=target时，左指针开始花钱，直到花到<target后，右指针再继续挣钱
        while right < len(nums):
            total += nums[right]
            while total >= target:
                min_len = min(min_len, right - left + 1)
                total -= nums[left]
                left += 1
            right += 1
        return 0 if min_len == 10**6 else min_len
```

Eg3. 找到字符串中所有字母异位词

给定两个字符串 `s` 和 `p`，找到 `s` 中所有 `p` 的 **异位词** 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

示例：

```
输入: s = "cbaebabacd", p = "abc"
输出: [0,6]
解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的异位词。
```

解：

可以使用滑动窗口+哈希表来进行解决：维护一个长度为`len(p)`的定长窗口以及一个哈希表，哈希表中记录当前窗口中每个字符的出现次数（一个子串中各个字符的出现次数和目标串相同，那么它就是目标串的异位词。所有异位词问题都可以以类似方式解决），每次移动一位后就比较一下当前窗口的哈希表和目标串的哈希表是否完全一样。注意，两个哈希表初始化时都要把'a'到'z'所有key都加入，否则比较的时候可能会受到某些计数为0但在一个表中存在而另一个表不存在的key的影响

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        if len(s) < len(p): return []

        hashmap = {}    # {'a':0, 'b':0, ...}
```

```
        hashmap_tmp = {} # {'a':0, 'b':0, ...}
        for i in range(97, 123):
            hashmap[chr(i)] = 0
            hashmap_tmp[chr(i)] = 0

        # construct reference hashmap
        for char in p:
            hashmap[char] += 1

        ret = []
        # 首次构建大小为n的定长窗口
        left, right = 0, 0
        for right in range(len(p)):
            hashmap_tmp[s[right]] += 1
        if hashmap == hashmap_tmp:
            ret.append(left)

        while right < len(s) - 1:
            hashmap_tmp[s[left]] -= 1
            left += 1
            right += 1
            hashmap_tmp[s[right]] += 1
            if hashmap == hashmap_tmp:
                ret.append(left)

        return ret
```

Eg4. 最小覆盖字串

给你一个字符串 `s` 、一个字符串 `t` 。返回 `s` 中涵盖 `t` 所有字符的最小子串。如果 `s` 中不存在涵盖 `t` 所有字符的子串，则返回空字符串 `""` 。

示例：

```
示例 1：
输入：s = "ADOBECODEBANC", t = "ABC"
输出："BANC"
解释：最小覆盖子串 "BANC" 包含来自字符串 t 的 'A'、'B' 和 'C'。

示例 2：
输入：s = "a", t = "a"
输出："a"
解释：整个字符串 s 是最小覆盖子串。

示例 3：
输入：s = "a", t = "aa"
输出：""
解释：t 中两个字符 'a' 均应包含在 s 的子串中，
因此没有符合条件的子字符串，返回空字符串。
```

解：

本题类似于”异位词“的进阶和推广版，只需窗口中目标串的每一个字符的数量都大于等于它在目标串中的数量（且其他字符数量随意），即可说明当前窗口是“有效覆盖子串”。因此，分别为当前滑动窗口和目标串准备一个字典`window_hashmap`和`t_hashmap`，且只记录目标串中出现过的各个字符的数量（因为其他字符数量多少都无关）。另外，维护一个变量`valid_num`记录当前窗口中有多少个t中的字符的数量恰好和t中的相等（或大于t中数量），用于判定当前窗口是否为有效覆盖字串，当`valid_num`等于t中不同字符数量时则说明当前是一个有效覆盖子串。

具体而言，不断向右移动右指针，每遇到一个t中的字符就将其记录到`window_hashmap`中，并查看此时该字符数量是否足够了，如果该字符数量足够了的话就将`valid_num+=1`。然后，看一下此时`valid_num`是否等于t中不同字符的数量，如果是的话说明当前窗口已经成为了一个有效覆盖子串，则开始尝试不断右移左指针来缩小滑动窗口（直到`valid_num`不再等于t中字符数量），不断丢弃左侧的字符以期找到更小的有效覆盖子串。若发现左侧离开窗口的是t中的字符，则需要把`window_hashmap`中记录的它的数量-1，并看一下此时窗口中该字符的数量是否还足够，如果发现已经不足了的话，则需要将`valid_num-=1`，说明此时已经不再是有效覆盖子串了。

```
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        if len(t) > len(s):
            return ""

        min_start = 0    # 记录当前找到的最小覆盖子串的起始位置
        min_len = 10**6  # 记录当前找到的最小覆盖字串的长度

        left, right = 0, 0  # 滑动窗口的左右指针

        t_hashmap = {}      # 记录t字符串中每个字符的数量
        window_hashmap = {}     # 记录当前窗口中每个t中字符的数量（不在t中的字符数量没有意义，不用记录）
        valid_num = 0    # 记录当前窗口中有多少个t中的字符的数量恰好和t中的相等（或大于t中数量），用于判定当前窗口是否为有效覆盖字串，当valid_num等于t中不同字符数量时则说明当前是一个有效覆盖子串

        # 然后把t_hashmap填充好
        for char in t:
            if char not in t_hashmap.keys():
                t_hashmap[char] = 1
            else:
                t_hashmap[char] += 1

        # 然后开始移动右指针
        while right < len(s):
            cur_char = s[right]
            right += 1

            # 若新加入的字符存在于t中，则将其添加到滑动窗口字典window_hashmap中
            if cur_char in t_hashmap.keys():
                if cur_char not in window_hashmap.keys():
                    window_hashmap[cur_char] = 1
                else:
                    window_hashmap[cur_char] += 1
                # 若添加了这个字符恰好使得窗口和t中该字符的数量相同了，则将valid_num+1
```

```
                if window_hashmap[cur_char] == t_hashmap[cur_char]:
                    valid_num += 1

            # 若此刻t中所有的字符都被窗口凑齐了，且数量也恰好都相等或大于t中数量，则此时滑动窗口中就是一个有效覆盖字串，则开始移动左指针缩小窗口，不断剔除没用的字符并更新最小覆盖字串大小，直到valid_num不再有效，此时可以看看是不是找到了一个更小的覆盖子串
            while valid_num == len(t_hashmap.keys()):
                # 如果发现此时的字串长度小于之前找到的最小值，则进行一次更新
                if right - left < min_len:
                    min_len = right - left
                    min_start = left

                # 然后继续将左指针右移一位，如果左边离开窗口的字母恰好在t中的话，则首先将其在window_hashmap中-1表示其离开了窗口，然后看下此时窗口中出现它的次数是否还大于等于t中，如果不是的话则说明窗口中该字母数量不足了，需要将valid_num-1
                left_char = s[left]
                if left_char in t_hashmap.keys():
                    window_hashmap[left_char] -= 1
                    if window_hashmap[left_char] < t_hashmap[left_char]:
                        valid_num -= 1
                left += 1

        return s[min_start: min_start + min_len] if min_len < 10**6 else ""
```

Eg5. 滑动窗口最大值

给你一个整数数组 `nums`，有一个大小为 `k` 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 `k` 个数字。滑动窗口每次只向右移动一位。

返回 *滑动窗口中的最大值* 。

示例：

```
输入：nums = [1,3,-1,-3,5,3,6,7], k = 3
输出：[3,3,5,5,6,7]
解释：
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

解：

本题有点类似“最小栈”问题，都是需要在O(1)复杂度内得到每个状态下的最值。和最小栈不同的是，本问题的滑动窗口本质上是一个先进先出的队列数据结构，因此考虑使用一个双端队列`max_queue = deque()`来维护窗口中最大值，想办法让队首元素总是当前窗口的最大值。在这个双端队列中，队首元素为当前窗口最大值，队列中保存的元素都是有可能成为未来窗口最大值的元素，且确保队列中元素值依次递减，以便让更大的元素优先成为未来窗口最大值，并及时将不可能成为最大元素的较小元素淘汰掉。

可以设计算法如下：令窗口从`[i, j] = [1-k, 0]`开始移动（一开始的时候不是完整窗口，不需要统计它们的最值，但必须让窗口右端从0开始移动以便正确构造队列）。

- 首先考虑离开窗口的左端元素`nums[i-1]`：若当前窗口是完整窗口，则其向右移动1位后，原窗口左端元素会离开窗口，如果它恰好也是队首元素的话，那么就需要将其从队列中弹出，以后的窗口不能再把它当作最大元素了
- 然后考虑即将加入窗口的右端元素`nums[j]`：首先，将其不断和当前队列中的队尾元素相比，然后逐个弹出每个比`nums[j]`小的元素，因为`nums[j]`不仅比它们大，未来还会比它们更晚离开窗口，因此`nums[j]`的加入使得这些元素不再有任何可能成为当前或未来某个窗口的最大值，将它们弹出队列。然后再将`nums[j]`加入到队尾，储备为未来窗口最大值的可能候选者（并保持队列的递减结构）
- 此时移动窗口给队列带来的操作已经全部完成。最后，如果此时的窗口是一个完整窗口，那么需要记录它的最大值，将此时队首元素加入结果列表即可。

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if len(nums) == 0: return nums[0]

        results = []

        from collections import deque
        max_queue = deque()      # 双端队列，保持队首一直是当前窗口中的最大元素，且队列中保持单调递减

        # 可以使用zip，来同时移动左右端点i,j
        for i, j in zip(range(1-k, len(nums)+1-k), range(0, len(nums))):
            # 如果移动窗口后，左端离开窗口的元素nums[i-1]恰好就是队列首元素，则以后的窗口用不上这个元素了，因此将队列首元素移除
            if i > 0 and max_queue[0] == nums[i-1]:
                max_queue.popleft()

            # 然后考虑右端即将加入窗口的元素nums[j]，从后往前依次弹出队列中所有比它小的元素
            # 因为nums[j]不仅比它们大，而且位置还在它们右边，将来会比它们晚离开队列，因此队列中nums[j]左侧的这些比它小的元素留着就没用了，从此以后窗口中最大的元素肯定不会是它们了，至少会是nums[j]
            while max_queue and max_queue[-1] < nums[j]:
                max_queue.pop()

            # 然后将nums[j]加入到队列中
            max_queue.append(nums[j])

            # 此时本轮窗口移动造成的队列变动就全都完成了，该收集当前窗口的最大值结果max_queue[0]了
            # 如果i>=0说明此时已经形成完整窗口，因此需要收集它的最大值
            if i >= 0:
                results.append(max_queue[0])
        return results
```

哈希表

一般用于快速通过“值”找到“index”，也即键:值一般就是`value:idx`

Eg1: 两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出 **和为目标值** *`target`* 的那 **两个** 整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案，并且你不能使用两次相同的元素。

你可以按任意顺序返回答案。

解析：为了能在一遍遍历中找到结果，因此使用一个哈希表来保存每个遍历过的value:index，这样在每遍历到一个新的元素时，就可以直接查找之前是否有可配对的值，并可以得到其相应的index来返回。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        # 哈希表：键、值分别为 value 和 idx，可以快速索引到之前已经看到的数字对应的idx
        mapping = dict()    # value: idx
        for i, num in enumerate(nums):  # 只需遍历1遍：每看到一个数字就在哈希表中寻找之前数字中有没有匹配的，如果没有就将当前数字存入哈希表
            if target - num in mapping.keys():  #
                return [i, mapping[target-num]]
            mapping[num] = i
        return []
```

Eg2. 和为K的子数组

给你一个整数数组 `nums` 和一个整数 `k` ，请你统计并返回 *该数组中和为 `k` 的子数组的个数* 。

子数组是数组中元素的连续非空序列。

解：

最直接的思路是通过移动两个指针来以$O(N^2)$的复杂度来暴力遍历所有可能的子数组。事实上，可以利用前缀和+哈希表，通过记录和复用已经遍历过的前缀，然后通过前缀之差来获得中间某段子数组的和，只遍历一遍即可统计到所有和为k的子数组。

本质上，将子数组之和转化为前缀之差，然后通过一遍遍历统计并记录各个前缀，在思路上就和“两数之和”很类似了。

具体而言，设置一个哈希表`prefix_count`，来记录每种可能的前缀和出现过的次数，也即`prefix_count[sum_]`表示前缀和为`sum_`的位置的数量（初始化时`prefix_count[0]=1`，空数组的前缀为0）。维护一个`current_sum`用于累加表示从数组开头到当前位置的前缀和，然后从头开始遍历数组的每个位置：每遍历到一个位置，首先累加出该位置的前缀和，然后看一下`current_sum - k`这个前缀和是否在`prefix_count.keys()`中出现过（以及出现了几次），如果出现过的话，则从那个位置到当前位置之间的子数组的和就是`k`（也即前缀和`current_sum`和前

缀和`current_sum-k`之差），此时就找到了`prefix_count[current_count-k]`个和为k的子数组，将其纳入全局数量统计即可

```
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        prefix_count = dict()   # prefix_count[sum_]表示前缀和为sum_的位置数量
        prefix_count[0] = 1     # 前缀和为0的子串数量初始化为1，也即什么都没有

        current_sum = 0     # 当前位置的前缀和（包括当前位置本身的值）
        count = 0       # 找到的和为k的子串数量

        for i in range(len(nums)):
            current_sum += nums[i]
            # 若前缀和为current_sum - k的位置也出现过，则从那个（些）位置到当前位置之间的子串就是和为k的子串，且这种子串共有prefix_count[current_sum-k]个，因为每个前缀和为current_sum-k的位置到当前位置都是一个和为k的子串
            if current_sum - k in prefix_count.keys():
                count += prefix_count[current_sum-k]

            # 将当前前缀和的统计数量+1
            if current_sum in prefix_count.keys():
                prefix_count[current_sum] += 1
            else:
                prefix_count[current_sum] = 1
        return count
```

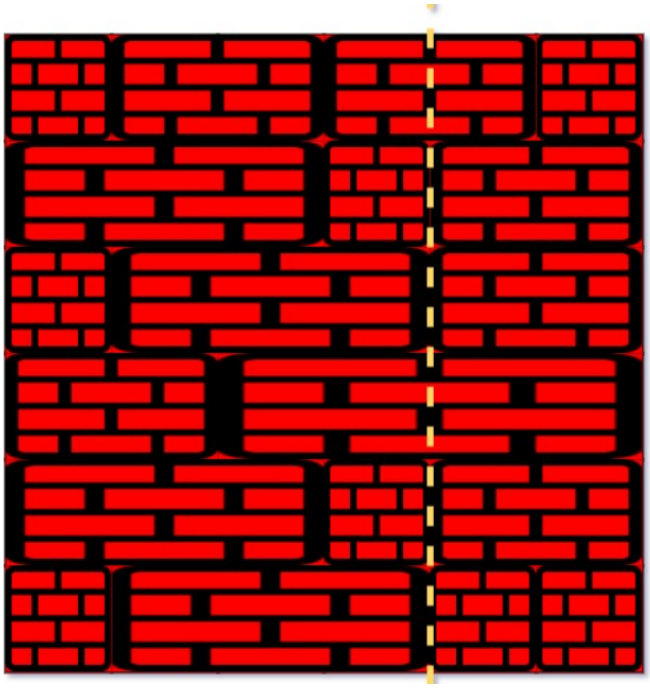
Eg 3. 砖墙

你的面前有一堵矩形的、由 `n` 行砖块组成的砖墙。这些砖块高度相同（也就是一个单位高）但是宽度不同。每一行砖块的宽度之和相等。

你现在要画一条 **自顶向下** 的、穿过 **最少** 砖块的垂线。如果你画的线只是从砖块的边缘经过，就不算穿过这块砖。**你不能沿着墙的两个垂直边缘之一画线，这样显然是没有穿过一块砖的。**

给你一个二维数组 `wall` ，该数组包含这堵墙的相关信息。其中，`wall[i]` 是一个代表从左至右每块砖的宽度的数组。你需要找出怎样画才能使这条线 **穿过的砖块数量最少** ，并且返回 **穿过的砖块数量** 。

示例：



```
输入：wall = [[1,2,2,1],[3,1,2],[1,3,2],[2,4],[3,1,2],[1,3,1,1]]
输出：2
```

解：

可以通过相反角度考虑问题：纵向穿过最少的砖，就相当于纵向穿过最多的缝隙，也即想要找到出现次数最多的缝隙位置（y方向/横向上的位置）。虽然每个砖的长度都不一样使得不好统计，但缝的位置很好确定：只需累加其所在行中前方的所有砖块长度即可。因此，维护一个哈希表，其键为出现过的缝隙的位置（y方向/横向），值为该位置的缝隙出现过的次数。依次遍历每一行，每遇到一个缝隙就将其统计值+1。最后，找到统计值最大的缝隙位置，沿着其向下走即为通过砖块最小的位置。

```python
class Solution(object):
    def leastBricks(self, wall):
        """
        :type wall: List[List[int]]
        :rtype: int
        """
        hashmap = {}
        # 依次遍历每一行
        for i in range(len(wall)):
            acc = 0 # 当前行中y方向的累加值
            for j in range(len(wall[i])-1): # 遍历该行每一个砖块，注意由于边缘不算，因此该行最后一个砖块不应被遍历到
                acc += wall[i][j]
                if acc not in hashmap.keys():
                    hashmap[acc] = 1 # 若该缝隙位置第一次出现，则将其出现次数设为1
                else:
                    hashmap[acc] += 1 # 若该缝隙位置不是第一次出现，则将其出现次数+1

        # （特殊情况）最终，如果没有任何缝隙，则需要通过所有砖块
        if not hashmap:
            return len(wall)
```

```
        max_edge_nums = max(hashmap.values())
        return len(wall) - max_edge_nums
```

Eg4. 多数元素

给定一个大小为 `n` 的数组 `nums` ，返回其中的多数元素。多数元素是指在数组中出现次数 **大于** `⌊ n/2 ⌋` 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

解：

可以使用一个哈希表，键为某个元素值，值为该元素值迄今为止出现的次数，从头到尾遍历一遍`nums`列表并实时更新迄今为止出现最多的元素即可

```
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        hashmap = {}
        majority_ele = 0
        majority_num = 0
        for i in range(len(nums)):
            if hashmap.get(nums[i]):
                hashmap[nums[i]] += 1
            else:
                hashmap[nums[i]] = 1
            if hashmap[nums[i]] > majority_num:
                majority_num = hashmap[nums[i]]
                majority_ele = nums[i]
        return majority_ele
```

Eg4. LRU缓存

请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。

实现 `LRUCache` 类：

- `LRUCache(int capacity)` 以 **正整数** 作为容量 `capacity` 初始化 LRU 缓存
- `int get(int key)` 如果关键字 `key` 存在于缓存中，则返回关键字的值，否则返回 `-1` 。
- `void put(int key, int value)` 如果关键字 `key` 已经存在，则变更其数据值 `value` ；如果不存在，则向缓存中插入该组 `key-value` 。如果插入操作导致关键字数量超过 `capacity` ，则应该 **逐出** 最久未使用的关键字。

函数 `get` 和 `put` 必须以 `O(1)` 的平均时间复杂度运行。

解：由于要求`get`、`put`都需要O(1)运行，因此使用哈希表（字典）来保存key, value。然而，由于还想需要实现最近最少使用缓存的逻辑，也即各个(key:value)的顺序是需要进行组织的，因此进一步使用双向链表来保存每个数据，以便方便地进行数据位置移动操作。

这样一来，字典中保存的即为(key:node)，也即：可以以O(1)的复杂度检索到数据对应的节点，一方面可以提取其value，另一方面可以通过链表操作来改变其位置。

```
# 链表节点，包含数据的key,value，以及前后节点
class LinkNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        # 哈希表，(key:node)
        self.cache = {}

        # 设置一个伪头部和伪尾部，方便处理边界的插入、删除
        self.head = LinkNode()
        self.tail = LinkNode()
        # 将头部尾部先相连
        self.head.next = self.tail
        self.tail.prev = self.head
        self.capacity = capacity
        self.size = 0


    def get(self, key: int) -> int:
        if key not in self.cache.keys():
            return -1

        # 提取数据值，将节点移动到链表首部，然后返回数据值
        node = self.cache[key]
        self.moveToHead(node)
        return node.value



    def put(self, key: int, value: int) -> None:
        if key not in self.cache.keys():
            # 如果要放入的key目前不在cache中，则为其创建一个链表节点、将该节点加入cache、并加到链表头
部
            node = LinkNode(key, value)
            self.cache[key] = node
            self.addToHead(node)

            # 当前cache的体积+1，然后判断是否溢出，若是则移除链表末尾节点，并将其在cache中也删除
            self.size += 1
            if self.size > self.capacity:
                tail_node = self.removeTail()
                del self.cache[tail_node.key]
                self.size -= 1
        else:
```

```
            # 如果要放入的key已经在cache中，则通过哈希表找到其对应节点，修改节点的value值，并移动到头
部
            node = self.cache[key]
            node.value = value
            self.moveToHead(node)

    # 各种所需链表操作，预先定义好以方便使用
    def addToHead(self, node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def removeNode(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def moveToHead(self, node):
        self.removeNode(node)
        self.addToHead(node)

    def removeTail(self):
        tail_node = self.tail.prev
        self.removeNode(tail_node)
        return tail_node    # 用于删除cache中的对应键值对


# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

Eg5. O(1) 时间插入、删除和获取随机元素

实现`RandomizedSet` 类：

- `RandomizedSet()` 初始化 `RandomizedSet` 对象
- `bool insert(int val)` 当元素 `val` 不存在时，向集合中插入该项，并返回 `true` ；否则，返回 `false` 。
- `bool remove(int val)` 当元素 `val` 存在时，从集合中移除该项，并返回 `true` ；否则，返回 `false` 。
- `int getRandom()` 随机返回现有集合中的一项（测试用例保证调用此方法时集合中至少存在一个元素）。每个元素应该有 **相同的概率** 被返回。

你必须实现类的所有函数，并满足每个函数的 **平均** 时间复杂度为 `O(1)` 。

解：

使用`set()`虽然能O(1)完成insert和remove，但其无法做到O(1)时间随机返回一个元素。因此，本题使用一个哈希表+一个变长数组来实现：数组用来存储数据元素，哈希表中的`键:值`分别是某个元素值和它在数组中的索引，也即可以O(1)地找到它在数组中的位置。

- 当插入一个元素时，若其此时不在哈希表中则将它加到数组末尾，并将它在数组中的位置也记录到哈希表中；

- 当删除一个元素时，如果直接找到它在数组中的位置并删除的话，会破坏数组连续性，使得后边所有元素都要往前挪动一位，导致复杂度不再是O(1)。因此，可以在删除元素时将数组末尾的元素挪动覆盖到删除的位置，同时更新这个末尾元素在哈希表中记录的索引信息，这样就可以确保后续绝大部分元素在数组中无需被移动，使得复杂度是O(1)
- 随机获取元素时，只需在当前数组长度范围内随机生成一个索引值，然后将其对应的数组元素返回即可

```
import random
class RandomizedSet:

    def __init__(self):
        self.nums = []  # 存放各个元素
        self.pos_dict = {}  # 存放各个"元素:位置"映射

    def insert(self, val: int) -> bool:
        if val not in self.pos_dict.keys():
            self.nums.append(val)
            self.pos_dict[val] = len(self.nums) - 1
            return True
        else:
            return False

    def remove(self, val: int) -> bool:
        if val not in self.pos_dict.keys():
            return False
        else:
            # 待删除元素val在nums中的位置
            val_idx = self.pos_dict[val]

            # 在该位置上覆盖nums末尾元素
            self.nums[val_idx] = self.nums[-1]
            # 更新nums末尾元素在pos_dict中的映射位置为当前新位置
            self.pos_dict[self.nums[-1]] = val_idx
            # 将nums的最后一个元素弹出，完成将最后一个元素移动到当前位置的操作
            self.nums.pop()

            # 删除待删除元素val在pos_dict中的映射关系
            del self.pos_dict[val]
            return True

    def getRandom(self) -> int:
        # 随机获得一个idx，并返回nums中它对应的元素
        rand_idx = random.randint(0, len(self.nums)-1)
        return self.nums[rand_idx]
```

题型：哈希集合

可以通过将列表等数据结构变成集合（`set`），来去重并用$O(1)$的时间复杂度查找某个元素是否在列表中（当对时间复杂度要求较高，空间复杂度要求较低时适用）

Eg1：最长连续序列

给定一个未排序的整数数组 `nums` ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 `O(n)` 的算法解决此问题。

示例：

```
输入：nums = [100,4,200,1,3,2]
输出：4
解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。
```

解：对于数组中的每个元素`x`，逐个遍历`x+1,x+2,x+3,...`是否也在数组中，即可得到以`x`开头的最长连续序列。为了去重以及用$O(1)$复杂度查找元素是否在其中，将列表转换成一个集合，然后再对集合中的元素进行遍历。

另外，还有一个可优化的点在于：如果数组中每个`x`都要遍历一遍`x+1,x+2,...`，则会造成较大开销。事实上，对于某个元素`x`来说，若`x-1`也在数组中，则以`x-1`开头的最长连续序列一定比`x`开头的最长连续序列长度多1，因此对于每个`x`可以先查一下`x-1`是否在数组中，如果`x-1`在数组中则可以直接跳过对于`x`开头的连续序列的遍历，因为它一定不是最长的，至少有一个`x-1`开头的序列比它长。

```python
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        if not nums: return 0

        hash_set = set(nums)
        longest_consec_len = 1

        for num in hash_set:
            if num - 1 in hash_set:
                continue
            cur = num + 1
            consec_len = 1
            while cur in hash_set:
                consec_len += 1
                longest_consec_len = max(longest_consec_len, consec_len)
                cur += 1

        return longest_consec_len
```

题型：原地哈希

由于哈希表本身需要占据O(n)的额外空间，因此当要求使用O(1)额外空间时不能直接自己显式地定义一个哈希字典或集合，此时可以利用原地哈希的思路，利用输入数组本身来保存信息

Eg1. 缺失的第一个正数

给你一个未排序的整数数组 `nums` ，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为 `O(n)` 并且只使用常数级别额外空间的解决方案。

示例 1：

```
输入：nums = [1,2,0]
输出：3
解释：范围 [1,2] 中的数字都在数组中。
```

示例 2：

```
输入：nums = [3,4,-1,1]
输出：2
解释：1 在数组中，但 2 没有。
```

解：

由于要求常数额外空间，因此不能额外建立一个set用于查找；由于要求时间复杂度O(n)，因此不能先排序然后二分查找（最快也是O(NlogN)）。因此，本题使用原地哈希的思路，利用题目输入的`nums`本身来保存信息。

首先，假设`len(nums)==n`，那么最小的缺失正数最多是n+1，因为如果`1~n`恰好填满了`nums`的话就是这种情况。因此，可以将问题简化为统计`1~n`是否在`nums`中。因此，可以尝试将每个在`1~n`范围内的`nums[i]`都移动到`nums[nums[i]-1]`，也即用`nums`数组的位置索引来保存元素值信息：排列后的`nums`形如`nums[0]=1, nums[1]=2, ...`，这样在排列后只需再从头到尾遍历一遍`nums`，看看什么时候第一次出现`nums[i] != i+1`的情况，那么这个`i+1`就是第一个缺失的正数了。如果发现`nums[0]~nums[n-1]`完全对应`1~n`，那么就返回`n+1`即可。

具体而言，每遍历到一个`nums[i]`时，如果它在关心范围内，那么首先将它和原先的`nums[nums[i]-1]`进行位置互换，这样就可以把`nums[i]`移动到`nums[nums[i]-1]`位置，并临时将原先的`nums[nums[i]-1]`元素暂存到`nums[i]`。然后，考虑这个新的`nums[i]`（也即原先的`nums[nums[i]-1]`），如果发现它恰好也到了它正确的位置，那就不用管它了；如果发现它在关心范围内且此时没处于正确位置，那么就继续对这个新的`nums[i]`做和上述相同的处理以便把它也放到正确的位置，如此不断处理新的`nums[i]`，直到发现它不处于关心范围内了或已经处于正确位置了，再退出循环，继续遍历`nums[i+1]`

```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        n = len(nums)

        # 遍历数组中每一个元素，将其放到应该在的位置
        for i in range(n):
            while nums[i] >= 1 and nums[i] <= n:     # 如果1<=nums[i]<=n的话，那么就需要对它负责，把它放到应该在的位置
```

```
                # 如果nums[i]和nums[nums[i]-1]本来就相等的话，说明nums[i]对应的元素已经放到正确位置了，不用管了
                if nums[i] == nums[nums[i] - 1]:
                    break

                # 将nums[i]移动到nums[nums[i]-1]，并把原先的nums[nums[i]-1]移动到nums[i]临时保存
                nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]
                # 如果发现此时的nums[i]（原先的nums[nums[i]-1]）恰好也放了正确的元素的话，就不用管它了
                if nums[i] == i + 1:
                    break
                # 如果发现此时的nums[i]（原先的nums[nums[i]-1]）此时的位置不对，那么就需要对它负责，继续将它放到它该处于的位置，因此继续循环

        # 原地哈希完成后，从头到尾依次遍历nums[0]~nums[n-1]，看看是不是对应1~n，如果某一位不对应那说明这一位就是第一个没出现的整数
        for i in range(n):
            if nums[i] != i + 1:
                return i + 1

        # 如果1~n都在的话，就返回n+1
        return n + 1
```

Eg2. 矩阵置零

给定一个 m x n 的矩阵，如果一个元素为 0 ，则将其所在行和列的所有元素都设为 0 。请使用 原地 算法，也即使用O(1)的空间复杂度。

解：

本题不算是和哈希有关的问题，但和上一题一样也是要求空间复杂度O(1)，这类题的思路通常都是利用给定的数据结构的一部分来存储信息。

如果不考虑空间复杂度的话，则可以首先遍历矩阵中的每个元素，并用两个set来记录一下含有0点行和列，然后再遍历一下各行各列并把相应行列置为0（注意不能一次遍历同时记录和置0，因为把一整行/列置0后会对其他交叉点行列是否含0造成影响）。

进一步，如果想不使用额外空间的话，就考虑使用matrix本身的一部分来存储各行各列的含0信息。具体而言，使用第0行和第0列来保存这个信息：当发现第`i`行存在0元素时设置`matrix[i][0]=0`，当发现第`j`列存在0元素时设置`matrix[0][j]=0`。而这样会破坏第0行和第0列本身的含0信息，因此要在上述操作前首先单独记录一下原始的第0行和第0列是否含0，最后再相应地处理它们

```
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        m, n = len(matrix), len(matrix[0])
        # 首先标记一下第0行和第0列分别是否有0，这样最后可以处理他俩
```

```
        first_row_has_zero = any(matrix[0][j] == 0 for j in range(n))
        first_col_has_zero = any(matrix[i][0] == 0 for i in range(m))

        # 然后使用第0行和第0列标记其他行列是否需要置零
        # matrix[i][0] = 0表示第i行有0元素
        # matrix[0][j] = 0表示第j行有0元素
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0
                    matrix[0][j] = 0

        # 根据标记，将含0的行列置零
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        # 最后再根据一开始的记录来对第0行和第0列置0或不置0
        if first_row_has_zero:
            for j in range(n):
                matrix[0][j] = 0
        if first_col_has_zero:
            for i in range(m):
                matrix[i][0] = 0
```

栈

先进后出，多用于有匹配关系的问题：和栈顶匹配则弹出栈顶，匹配一个就扔（消掉）一个

Eg1：括号匹配

给一个只含有`[](){}`的字符串，判断其各类型括号是否都左右封闭

解：遇到左括号则入栈，遇到右括号则看其是否和栈顶顶左括号匹配，如果不匹配或此时栈空则说明这里不封闭，返回False。最终栈应被清空，如果还剩余左括号没被匹配的话也不封闭。

```
class Solution:
    def isValid(self, s: str) -> bool:
        # 如果长度是奇数则必然不行
        if len(s) % 2 == 1:
            return False

        # 左括号:
        left = ['(', '{', '[']
        # 右括号:
        right = [')', '}', ']']
        # 左右括号对应：因为是根据到来的右括号寻找栈顶的左括号，因此右括号为键
```

```
        pairs = {
            ')': '(',
            ']': '[',
            '}': '{'
        }

        stack = []
        for ch in s:
            if ch in left:  # 如果是左括号，则入栈
                stack.append(ch)
            else:   # 如是右括号，则看看栈顶是不是匹配的左括号，如果是则配对并出栈，否则说明不匹配(或栈空了)
                if not stack or stack[-1] != pairs[ch]:
                    return False
                else:
                    stack.pop()

        # 如果最后栈空了说明都成功匹配
        return not stack
```

Eg2. 最长有效括号

给你一个只包含 `'('` 和 `')'` 的字符串，找出最长有效（格式正确且连续）括号子串的长度。

示例：

```
输入：s = "(()"
输出：2
解释：最长有效括号子串是 "()"

输入：s = ")()())"
输出：4
解释：最长有效括号子串是 "()()"
```

解：

设置一个栈，其保存括号的索引index，其中：

- 栈底元素保持为当前已遍历元素中“最后一个未被匹配的右括号”的索引，将其作为一个“基准”
- 栈中其他元素为“基准”右括号右边的各个左括号的索引，也即当前基准后边所有等待被匹配的左括号索引

设置栈底右括号可以作为“基准”，来隔离掉其左边的子串（这个未匹配右括号的存在注定了其左边子串没有多余的左括号了，因此不可能再和后边的右括号进行匹配了），从其右边开始都是“有效”的左括号，是可以用于构成未来的完整有效串的。

每遍历到一个新的左括号就将其索引入栈，等待被匹配；

每遍历到一个新的右括号时：

- 如果它能被当前有效子串中的左括号匹配掉（体现为栈顶元素是个左括号，也即`stack.pop()`后栈还没空），那么用该新右括号的索引减去（弹出原左括号栈顶后的）新栈顶的索引就是以当前右括号为结尾的有效子串的最大长度，用其更新`max_len`。
- 而如果发现没有左括号可以跟它匹配了（体现为栈顶元素是个右括号，也即`stack.pop()`后栈空了），那么就说明它左边不再有多余的左括号可能和未来的右括号匹配了，将它自己作为新的基准元素压入栈底。

初始化栈时，在栈中放入占位符-1，防止第1个括号就是左括号'('导致和上述定义的冲突

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        # 栈底元素总是当前已遍历元素中的最后一个未被匹配的右括号的索引，其余元素都是左括号索引
        # 设置栈底右括号可以作为“基准”，来隔离掉其左边的子串（这个未匹配右括号的存在注定了其左边子串没有多余的左括号了，因此不可能再和后边的右括号进行匹配了），从其右边开始都是“有效”的子串，
        # 之后每遍历到一个新的右括号只要它能被当前有效子串中的左括号匹配掉，那么用该新右括号的索引减去弹出左括号后的新栈顶索引就是以当前右括号为结尾的有效子串的最大长度，用其更新max_len
        # 如果新加入某个右括号时，发现没有左括号可以跟它匹配了，那么就将它自己作为新的基准元素压入栈底
        # 初始时在栈中放入占位符-1，防止第1个括号就是左括号'('导致和上述定义的冲突
        stack = [-1]
        max_len = 0
        for i in range(len(s)):
            if s[i] == '(':     # 对于每个左括号'('，将其索引放入栈中等待被匹配
                stack.append(i)
            elif s[i] == ')':   # 对于每个右括号')'，将栈顶元素弹出
                stack.pop()
                if not stack:   # 如果弹出栈顶元素后栈空了，则说明弹出的是栈底元素，也即“基准”右括号的索引，那么说明当前遍历到的右括号没能找到前边的左括号匹配，因此把它自己放入栈中作为新的“基准”左括号
                    stack.append(i)
                else:   # 如果弹出栈顶元素后栈还没空，说明弹出的是个左括号，它可以和当前右括号匹配掉。
                    max_len = max(max_len, i - stack[-1])
        return max_len
```

Eg3：字符串解码

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: `k[encoded_string]`，表示其中方括号内部的 `encoded_string` 正好重复 `k` 次。注意 `k` 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 `k` ，例如不会出现像 `3a` 或 `2[4]` 的输入。

示例 1：

```
输入：s = "3[a]2[bc]"
输出："aaabcbc"
```

示例 2：

```
输入：s = "3[a2[c]]"
输出："accaccacc"
```

解：

使用栈来解决这种存在嵌套和匹配的问题。具体而言，设置一个`cur_num`来构造当前的倍数（考虑多位数的情况），设置一个`cur_str`来保存当前括号中正在构造的字符串，设置`stack`来保存当前已构造完成的字符串以及其所在括号对应的重复倍数。遍历字符串过程中，若遇到数字则更新`cur_num`；若遇到字母则直接加到当前正在构造的字符串`cur_str`上；若遇到`'['`则说明上一级括号下的第一级字符串已经构造完成，因此入栈上一级括号内的第一级字符串`cur_str`以及本括号对应的倍数`cur_num`，并将`cur_str`和`cur_num`重置；若遇到`']'`则说明当前括号构造完成，此时`cur_str`即为当前括号内的字符串，将其乘以本括号对应的倍数（通过出栈得到），并连接到上一级字符串（通过出栈得到）

```python
class Solution:
    def decodeString(self, s: str) -> str:
        stack = []      # 之前的字符串以及之前的倍数
        cur_str = ""    # 当前括号中正在构造的字符串
        cur_num = 0     # 当前正在构造的倍数，用于处理多位数倍数的情况

        for char in s:
            if char.isdigit():  # 如果当前是数字，则将其加到当前正在构造的倍数上
                cur_num = cur_num * 10 + int(char) # 处理多位数的情况
            elif char == '[':   # 左括号，说明又开启了一个新括号区域，则将当前已有字符串（该括号之前构建好的字符串）和该括号的倍数入栈
                stack.append((cur_str, cur_num))
                # 然后将当前字符串和当前倍数重置，为内层的下一个括号做准备
                cur_num = 0
                cur_str = ""
            elif char == ']':   # 右括号，说明当前括号区域结束，则弹出当前括号对应的倍数和之前已有字符串
                prev_str, num = stack.pop()
                cur_str = prev_str + cur_str * num  # 并将当前括号中构造好的字符串乘以倍数后加到之前弹出的字符串上，作为新的cur_str
            else:   # 普通字母，直接加到当前正在构造的字符串上
                cur_str += char
        return cur_str
```

示例：`3[a2[c]]`

```
'3': cur_num=3
'[': stack=[("", 3)]; cur_num=0; cur_str=""
'a': cur_str='a'
'2': cur_num=2
'[': stack=[("", 3), ('a', 2)]; cur_num=0; cur_str=""
'c': cur_str='c'
']': prev_str,num='a',2; cur_str='a'+2*'c'='acc'
']': prev_str,num="",3; cur_str=""+3*'acc'='accaccacc'
```

Eg4. 反转每对括号间的子串

给出一个字符串 `s`（仅含有小写英文字母和括号）。

请你按照从括号内到外的顺序，逐层反转每对匹配括号中的字符串，并返回最终的结果。（保证所有括号成对存在）

注意，您的结果中 **不应** 包含任何括号。

示例：

```
输入：(ab(cd)ef)()
输出：fecdba
注：由于cd被两层括号包括，所以对它的操作是反转一次后再反转一次，所以最终结果中它还是cd
```

解：

方法1：仿照字符串解码的思路

本题与字符串解码类似，也是括号和字符混搭的问题，会对于括号中的字符串做某种操作，且嵌套的括号也会作用。在这种问题中，入栈的不再是括号本身，而是当前构造的字符串状态（上题中还包括了倍数状态）。套路为：

- 维护一个栈和一个当前构造的字符串状态
- 遍历原字符串：
  - 如果遇到左括号，则说明开启了一个新区域，则将当前构造的字符串状态入栈，然后重置当前字符串状态
  - 如果遇到右括号，则说明当前区域构造完成，此时一方面对当前字符串进行规定操作（如上题中的重复、本题中的翻转），另一方面弹出栈顶的（上一层的）字符串状态，然后将二者拼接为当前构造的字符串即可（它会在遇到新的左括号时再次入栈，因此这里不需要进一步做任何操作了）
  - 如果遇到普通字符，则更新当前字符串状态即可
- 最终返回遍历结束时的当前字符串即可

```python
class Solution:
    def reverseParentheses(self, s: str) -> str:
        stack = []
        cur_str = ""    # 当前字符串
        for char in s:
            if char == '(': # 如果是左括号，则说明开启了一个新区域，将当前字符串（作为一个整体）入栈，并清空当前字符串
                stack.append(cur_str)
                cur_str = ""
            elif char == ')':   # 如果是右括号，则说明当前区域结束了，要对当前字符串做一次翻转，然后将其接到上一个构造出来的字符串上
                cur_str = cur_str[::-1]
                prev_str = stack.pop()
                cur_str = prev_str + cur_str

            else:   # 如果是普通字符，则加到当前字符串上
                cur_str += char
```

```
        return cur_str
```

方法2：如果不保证所有括号都能匹配，在最终结果中忽略不匹配的括号

此时因为括号不保证都能匹配，因此需要把括号也入栈用于做匹配，和标准做法不太一样。

将左括号和字符都入栈，每次遇到右括号时，则不断弹出栈顶直到遇到匹配的左括号（或到达栈底）：如果遇到了匹配的左括号，则将刚才弹出的这段子串进行反转，然后放回栈中用于以后的处理（也即，将括号产生的影响作用到其包括的字符串上，然后这对括号就可以扔掉了）；如果到达栈底也没遇到匹配的左括号，那么说明这个右括号是多余的，则还把这段字符串原封不动地放入栈中。最后，栈中剩余的就是处理好顺序的所有字符以及多余的括号，只需把多余括号删掉即可得到处理后的最终字符串。

例如：

```
input = (ab(cd)ef)()
['(']
['(', 'a']
['(', 'a', 'b']
['(', 'a', 'b', '(']
['(', 'a', 'b', '(', 'c']
['(', 'a', 'b', '(', 'c', 'd']
# 遇到右括号，弹出字符直到左括号
['(', 'a', 'b', '('], tmp_str = 'cd'
# 再将翻转后的字符串压回栈里
['(', 'a', 'b', 'd', 'c']
# 继续
```

```
class Solution:
def reverseParentheses(self, s: str) -> str:
  stack = []
  for char in s:
      if char != ')':      # 左括号或字符，将其入栈
          stack.append(char)
      else:        # 右括号
          tmp_seq_ls = []
          while stack and stack[-1] != '(':     # 依次弹出当前右括号左边的字符，直到遇到左括号或栈底
              tmp_seq_ls.append(stack.pop())  # 加入临时子串的顺序直接就是翻转后的顺序
          if not stack:        # 如果没有左括号可以和这个右括号匹配，说明这个右括号是多余的，则它左边弹出的这些字符不应该被翻转
              tmp_seq_ls = tmp_seq_ls[::-1]
          else:
              stack.pop()      # 把对应的左括号弹出
          # 将翻转（或不翻转）后的这部分子串放回栈中，以便如果以后也有括号的话还能继续被处理
          stack.extend(tmp_seq_ls)

  # 最后，栈中只剩下处理好顺序的字符串以及某些没能被匹配的左括号，则将多余括号去除后剩下的就是处理好顺序的输出字符串
```

```
    result = ''.join(stack).replace('(', '')
    return result
```

Eg5. 基本计算器

给你一个字符串表达式 `s` ， `s` 由数字、`'+'`、`'-'`、`'('`、`')'`、和 `' '` 组成，请你实现一个基本计算器来计算并返回它的值。

示例：

```
输入：s = " 2-1 + 2 "
输出：3

输入：s = "(1+(4+5+2)-3)+(6+8)"
输出：23
```

解：

和前两题类似，维护一个`cur_num`表示当前正在构造的数值，`cur_res`表示当前层级下正在构造的累加（减）结果，`cur_sign`表示当前正在构造的数值前边的正负号。`(`和`)`的处理逻辑和前两题基本一样。

```
class Solution:
    def calculate(self, s: str) -> int:
        stack = []
        cur_num = 0     # 当前正在构造的单个数值
        cur_res = 0     # 当前括号层级下正在构造的累加（减）结果
        cur_sign = 1    # 当前正在构造的数值前边的正负号

        for ch in s:
            # 如果是数字，则加到当前构造的数值上
            if ch.isdigit():
                cur_num = cur_num * 10 + int(ch)
            # 如果是+号，则说明当前数值cur_num已构造完毕，可以将其乘以其前边的符号并累加到cur_res上，然后重置cur_num并更新符号为1，准备为下一个数值的符号
            elif ch == '+':
                cur_res += cur_sign * cur_num
                cur_sign = 1
                cur_num = 0
            # 如果是-号，同理
            elif ch == '-':
                cur_res += cur_sign * cur_num
                cur_sign = -1
                cur_num = 0
            # 如果是'('，说明当前层级暂时构造完毕，因此将当前层级结果cur_res和当前符号（为下一个层级准备的）cur_sign入栈，并重置cur_res和cur_sign
            elif ch == '(':
```

```
                stack.append((cur_res, cur_sign))
                cur_res = 0
                cur_sign = 1
            # 如果是')'，一方面说明当前数值cur_num构造完毕，同时也说明当前层级cur_res彻底构造完毕，将当前数值cur_num重置后，可以从栈中弹出之前保存的cur_res对应的符号，以及上一层构造结果，并将二者进行融合，作为新的cur_res
            elif ch == ')':
                # 当前num构造完毕，将其累加到cur_res上并重置num
                cur_res += cur_sign * cur_num
                cur_num = 0
                # 然后将当前层级的符号和上一层的结果出栈，并将其和当前cur_res进行融合，得到新的cur_res
                prev_res, prev_sign = stack.pop()
                cur_res = prev_res + prev_sign * cur_res
            else:
                continue

        # 将最后一个数字也加进来（如果最后一个字符是括号也没关系，因为此时cur_num=0，不影响结果）
        cur_res += cur_sign * cur_num
        return cur_res
```

Eg6. 最小栈（辅助栈）

设计一个支持 `push` ， `pop` ， `top` 操作，并能在常数时间内检索到最小元素的栈。

实现 `MinStack` 类:

- `MinStack()` 初始化堆栈对象。
- `void push(int val)` 将元素val推入堆栈。
- `void pop()` 删除堆栈顶部的元素。
- `int top()` 获取堆栈顶部的元素。
- `int getMin()` 获取堆栈中的最小元素。

解：除了栈本身以外，同时再设置一个辅助栈（最小栈），其中的每个元素对应主栈中相应元素入栈后，栈中的最小元素（也即可以看成一个保存“状态”的栈，其保存了每个主栈元素对应的最小元素）。这样一来，任何时刻主栈中的最小值都对应最小栈的栈顶

具体操作：

- 入栈时：一方面将`val`入主栈，另一方面将`min(val, min_stack[-1])`入最小栈（也即当前元素入栈后，栈中的最小值）
- 出栈时：同时将主栈顶和最小栈顶弹出

```
class MinStack(object):
```

```
    def __init__(self):
        self.stack = []
        self.min_stack = []

    def push(self, val):
        """
        :type val: int
        :rtype: None
        """
        self.stack.append(val)
        if not self.min_stack:
            self.min_stack.append(val)
        else:
            self.min_stack.append(min(val, self.min_stack[-1]))

    def pop(self):
        """
        :rtype: None
        """
        self.stack.pop()
        self.min_stack.pop()

    def top(self):
        """
        :rtype: int
        """
        return self.stack[-1]

    def getMin(self):
        """
        :rtype: int
        """
        return self.min_stack[-1]
```

题型：单调栈

多用于存在“大小关系”、“配对-消去”的问题。一般来说栈里保存的是索引idx，需要用的时候可以随时找到相应的值

Eg1. 单调栈：每日温度

给定一个整数数组 `temperatures` ，表示每天的温度，返回一个数组 `answer` ，其中 `answer[i]` 是指对于第 `i` 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 `0` 来代替。

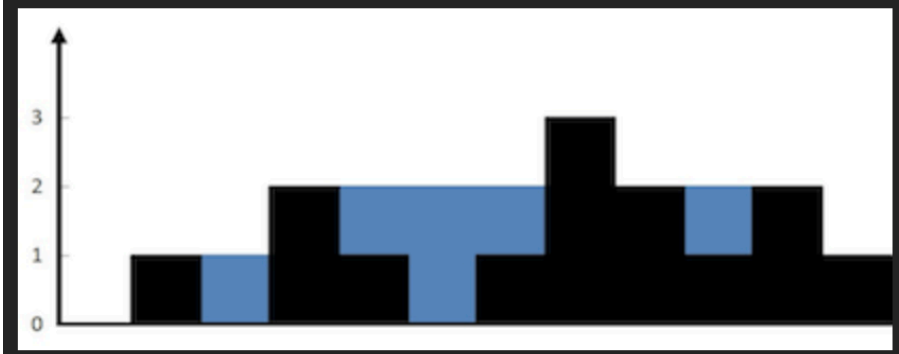
解析：在一遍遍历中，为了每遍历到一个新温度就能干掉其之前所有比其更低的温度（也即填上这些更低温度的 `answer` 值），维护一个单调栈保存各个遍历过且尚未见到更高温度的天的索引值：从栈底到栈顶温度依次降低。每遍历到一个值时，若栈空则入栈，否则比较其与栈顶，并不断弹出比他小的栈顶，这些栈顶的 `answer` 值也就被设为当前索引与它们索引之差，最后再将当前索引入栈，等待后续更高的温度。

```python
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        if len(temperatures) == 0:
            return 0

        ans = [0] * len(temperatures)
        stack = []  # 单调栈，保存当前尚未找到后续更高温度天的那些天的idx。栈底为当前温度最高的天的idx，栈顶为当前温度最低的天的idx，温度依次递减

        for i in range(len(temperatures)):
            if not stack:
                stack.append(i) # 如果栈空，则将当前idx入栈
            else:
                while stack and (temperatures[stack[-1]] < temperatures[i]):  # 若当前idx的温度大于栈顶，则栈顶对应的天的后续更高温度天即为当前idx，将栈顶弹出并设置其ans值为当前idx。
                # 不断弹出比当前idx温度更小的天，直到剩下的都是比它温度高的
                    top_idx = stack.pop()
                    ans[top_idx] = i - top_idx
                stack.append(i)
        return ans
```

Eg2：单调栈：接雨水

给定 `n` 个非负整数表示每个宽度为 `1` 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：



> **输入：** height = [0,1,0,2,1,0,1,3,2,1,2,1]
> **输出：** 6
> **解释：** 上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = 0
        stack = []  # 单调栈，从底到顶的idx对应的高度逐渐减小
```

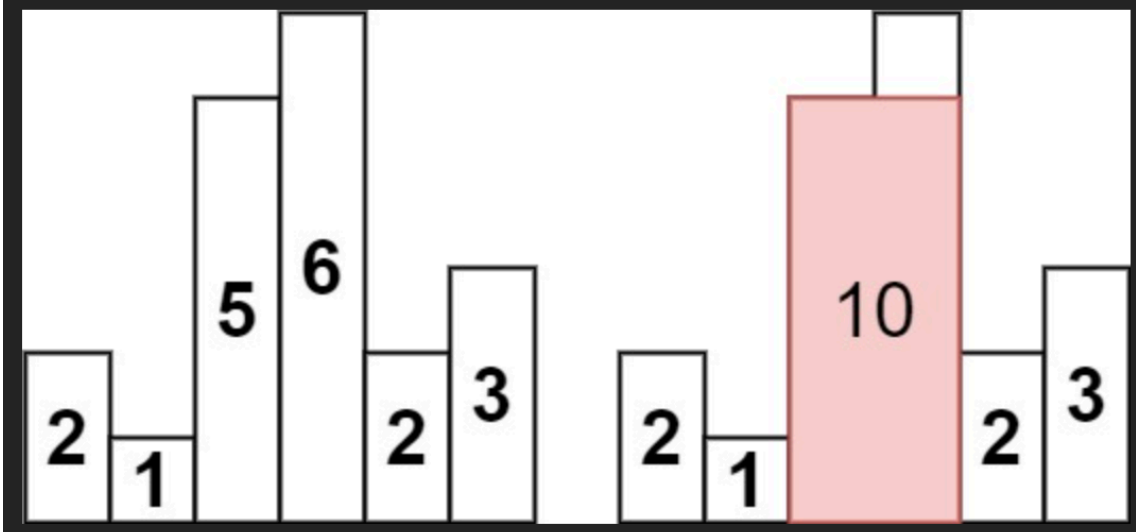
```
        # 一次遍历即可
        for i, hei in enumerate(height):
            # 当栈不空且有至少2个元素时，若当前高度大于栈顶，由于栈顶的下一个元素必然也高于栈顶，因此出现了水槽，可以统计一部分水的体积，然后干掉当前栈顶。如此迭代直到不再能构成水槽。
            while stack and hei > height[stack[-1]]:
                top_idx = stack.pop() # 弹出栈顶
                if not stack:   # 如果栈中只有1个元素，则不能构成水槽
                    break
                left_idx = stack[-1] # 水槽左侧
                cur_width = i - left_idx - 1 # 水槽宽度
                cur_height = min(height[left_idx], height[i]) - height[top_idx] # 水槽高度
                ans += cur_width * cur_height #在总水量中加上当前水槽
            stack.append(i) # 干掉左侧的所有水槽后，将当前idx入栈
        return ans
```

Eg3. 柱状图中最大的矩形

给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。

示例 1:



输入: heights = [2,1,5,6,2,3]
输出: 10
解释: 最大的矩形为图中红色区域，面积为 10

解：

本题和接雨水是“反过来”的：接雨水统计的是柱子间的空缺面积，而本题统计的是柱子覆盖的面积。

在本题中，最终得到的最大面积矩形的高度一定是某个柱子的高度，因此可以把问题转化成求取以各个柱子为高的最大矩形，然后再从中取最大的一个即可。而对于某个柱子而言，决定以它为高度的最大矩形的宽度的因素是：它左侧第一个比它低的柱子的位置 和 它右侧第一个比它低的柱子的位置，因为从本柱子出发，向左直到左侧第一个比它低的柱子之前都是比它高的柱子，都是可以用于构成以当前柱子为最大高度的矩形的，右侧同理。

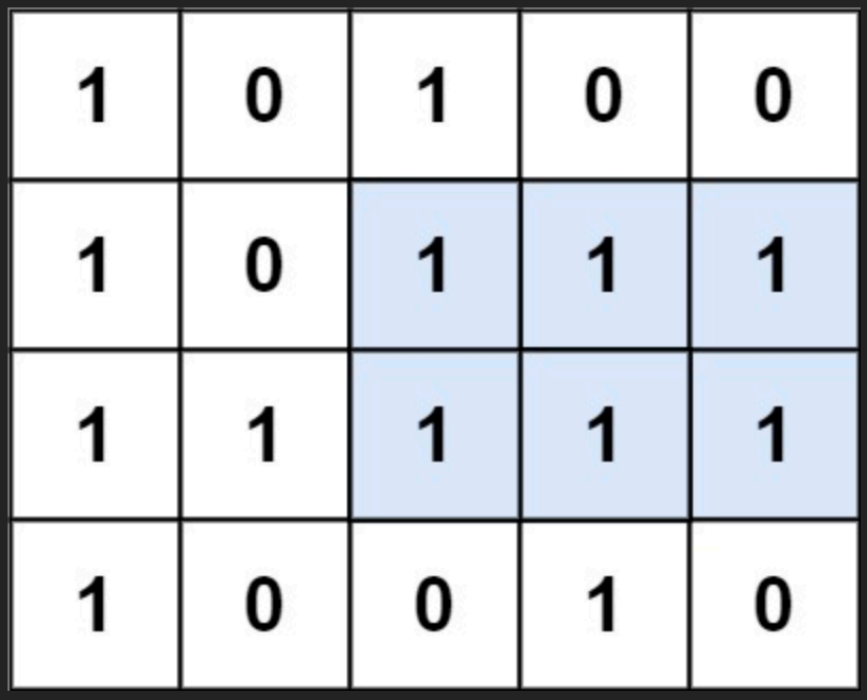
在从左到右遍历柱子的过程中，比较容易找到左侧第一个比它低的柱子，但右侧第一个比它低的柱子属于未来信息，当遍历到当前柱子时是无法确定的。因此可以参考“每日温度”的思路中对于未来信息的处理：维护一个单调栈，从栈底到栈顶保存的索引对应的柱子高度依次变高，也即栈中每个柱子左侧第一个比它矮的柱子的索引就在它的下一位，栈中的每个柱子都在等待找到它右侧第一个比它更矮的柱子。构建栈道过程中，从头到尾依次遍历各个柱子，每遍历到一个柱子时，就依次干掉当前栈中各个比它高的栈顶——当前柱子就是这些栈顶柱子右侧第一个更矮的柱子。每弹出一个栈顶，就可以通过它右侧第一个更矮的柱子的索引（也即当前遍历到的柱子）减去它左侧第一个更矮的柱子的索引（也即新栈顶）来获得最大矩形的宽度，然后乘以它的高度，就得到了以它为高度的最大矩形的面积。

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        stack = []  # 单调栈，从底到顶依次变高，这样每次栈顶柱子左侧的第一个比它低的柱子在栈中也和它相邻
        max_area = 0
        heights = [0] + heights + [0]  # 添加左右边界便于处理
        n = len(heights)

        for i in range(n):  # 遍历每个柱子
            # 依次干掉栈中各个比当前柱子高的柱子：当前柱子就是它右边第一个比它矮的柱子，弹出后的新栈顶stack[-1]就是它左边第一个比它矮的柱子
            while stack and heights[i] < heights[stack[-1]]:
                h = heights[stack.pop()]    # 弹出的柱子的高度
                left = stack[-1]            # 新栈顶，也即弹出柱子左侧第一个比它矮的柱子
                width = i - left - 1    # 以弹出柱子为高的最大矩形的宽度：弹出柱子右侧第一个比它矮的柱子的索引 - 左侧第一个比它矮的柱子的索引（也即当前柱子） - 1
                area = h * width    # 以弹出柱子为高的最大矩形的面积
                if area > max_area:
                    max_area = area     # 更新全局的最大矩形面积
            stack.append(i)     # 干掉所有以当前柱子为右边界的柱子后，将当前柱子入栈

        return max_area
```

Eg4. 最大矩形

给定一个仅包含 `0` 和 `1` 、大小为 `rows x cols` 的二维二进制矩阵，找出只包含 `1` 的最大矩形，并返回其面积。

示例 1：

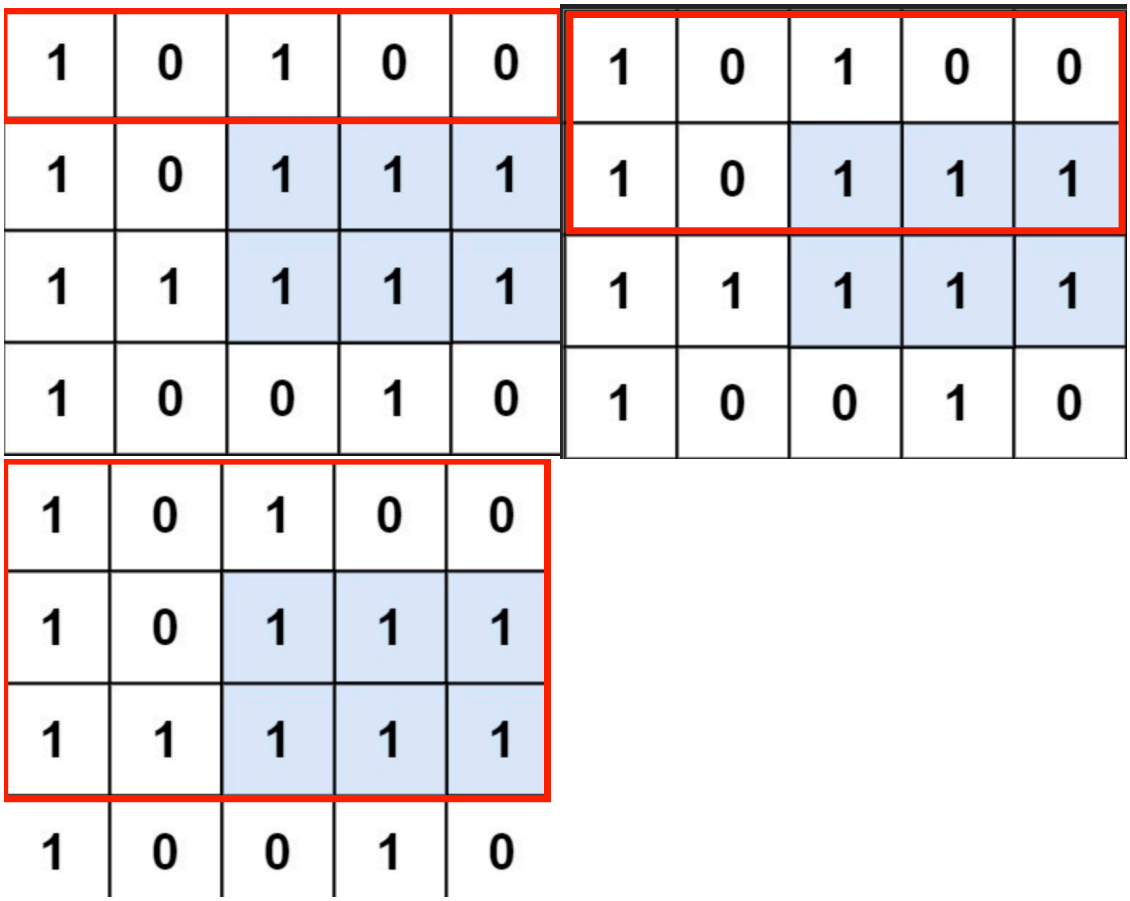



```
输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],
["1","1","1","1","1"],["1","0","0","1","0"]]
输出：6
```

解：

本题乍一看与“最大正方形”很相似，但由于矩形的长宽不一定相等，因此本题完全无法套用最大正方形那道题的dp做法。

事实上，本题可以通过直接对上一题“柱状图中的最大矩形”的方法进行扩展来求解：从上到下依次访问每一行，并利用上题解法来求出以该行为底的（在该行上方区域的）柱状图中的最大矩形面积：




```python
class Solution:
    def largestRectangleArea(self, heights):
        stack = []
        max_area = 0
        heights = [0] + heights + [0]
        for i in range(len(heights)):
            while stack and heights[i] < heights[stack[-1]]:
                h = heights[stack.pop()]
                left = stack[-1]
                width = i - left - 1
                max_area = max(max_area, h * width)
            stack.append(i)
        return max_area

    def maximalRectangle(self, matrix: List[List[str]]) -> int:
        if not matrix:
            return 0
        rows, cols = len(matrix), len(matrix[0])
        heights = [0] * cols
        max_area = 0

        for i in range(rows):    # 从上往下依次遍历每一行
            # 首先，算出以该行为底的柱状图高度分布
            # 如果该行某个位置为0，则该位置本次不可能对全局最大面积产生新贡献，因此将该位置的柱子高度设为0
            # 如果该行某个位置为1，则将该位置的柱子高度在上一轮的基础上+1
            for j in range(cols):
```

```
                heights[j] = heights[j] + 1 if matrix[i][j] == '1' else 0
            # 然后，调用单调栈算法来求出以该行为底的柱状图中的最大矩形面积
            max_area_row = self.largestRectangleArea(heights)
            # 最后，更新全局最大矩形面积
            max_area = max(max_area, max_area_row)
        return max_area
```

堆（优先队列）

Eg 1 数组中的第K个最大元素

给定整数数组 `nums` 和整数 `k`，请返回数组中第 `k` 个最大的元素。

请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

你必须设计并实现时间复杂度为 `O(n)` 的算法解决此问题。

解：可以使用堆来解决。维护一个大小为`k`的最小堆，确保堆顶永远是当前最小的元素。从头到尾依次遍历数组，将数组元素入堆，当堆大小到达`k`以后，再有元素入堆时就同时弹出堆顶（当前最小元素），这样就可以保证堆中总是当前遍历过的所有元素中最大的k个。最终只需取堆顶，就是整个数组中第k大的元素。

```
import heapq

def findKthLargest(nums, k):
    # 初始化一个空的最小堆
    min_heap = []

    # 遍历数组中的每个数字
    for num in nums:
        # 将数字加入堆中
        heapq.heappush(min_heap, num)

        # 如果堆的大小超过 k，则弹出堆顶最小值
        if len(min_heap) > k:
            heapq.heappop(min_heap)

    # 堆顶元素即为第 k 个最大的元素
    return min_heap[0]
```

Eg2. 前k个高频元素

给你一个整数数组 `nums` 和一个整数 `k` ，请你返回其中出现频率前 `k` 高的元素。你可以按 任意顺序 返回答案。

示例:

```
输入：nums = [1,1,1,2,2,3], k = 2
输出：[1,2]
```

解：

首先遍历一遍数组，构造一个哈希表来存储每个元素及其频数。然后，针对频数数组，问题就转化成了“求前k大的元素”，topk问题使用小顶堆解决即可：

```python
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # 首先构造一个哈希表，记录每个元素出现的频数
        freq_dict = {}
        for num in nums:
            if num not in freq_dict.keys():
                freq_dict[num] = 1
            else:
                freq_dict[num] += 1

        # 此时问题转化成：求频数数组中前k大堆值
        # 只需使用一个最大容量为k的小顶堆即可：当堆未满时，可以直接插入；当堆满时，插入一个元素后还需弹出堆顶
        # 这样就能保证：在当前遍历过的所有元素中，堆里的始终是前k大堆元素
        import heapq
        heap = []

        for num, freq in freq_dict.items():
            heapq.heappush(heap, (freq, num))   # 堆内部按照freq大小来排序
            if len(heap) > k:
                heapq.heappop(heap)

        # 最终留在堆里的就是频数前k大堆元素
        return [num for freq, num in heap]
```

Eg3. 合并K个升序链表

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表。

示例：

```
输入：lists = [[1,4,5],[1,3,4],[2,6]]
输出：[1,1,2,3,4,4,5,6]
解释：链表数组如下：
[
  1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6
```

解：

本题如果通过重复使用双指针来不断合并每两个链表的话会比较麻烦。事实上，这个情境下可以考虑最小堆（优先队列）：对于结果链表，每次都希望从剩余的所有节点中挑出最小的那个接在它后边，如此不断添加节点从而构造出结果链表。而最小堆的功能恰好就是每次都自动把当前堆中最小的节点浮到堆顶，因此只需把所有链表节点全都扔到堆中，然后不断取堆顶并将其连接到结果链表中即可。

为了优化空间占用（如果链表节点数量过多的话，全都扔到堆中就会造成空间爆炸），考虑缩小每个时刻堆中存的元素，尽可能确保堆中存的只是那些有可能作为下一个节点的节点。具体而言，考虑结果链表中的第一个节点（也即全局最小节点），它一定来自某个链表的head，因此初始化heap时可以只加入所有的head节点，因为只有head节点有可能作为这一步的下一个节点；对于后续步骤来说，每次弹出一个节点（也即当前最小节点），则它在原链表中的下一个节点（比它大一点）也成为了结果链表中可能的下一个节点，因此每一步都弹出堆顶并将其连接到结果链表上，然后将堆顶在原链表中的下一个节点加入堆中

```python
class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:
        # 给ListNode对象添加比较大小的逻辑，从而能使用heap进行排序
        ListNode.__lt__ = lambda node1, node2: node1.val < node2.val

        import heapq
        # 最小堆中保存了可能作为下一个最小节点的所有候选节点，其内部自动排序后，堆顶总是下一个最小的候选节点
        # 初始时，将列表中所有非空的链表头节点加入最小堆中，它们作为结果链表中第一个节点的候选者
        least_heap = [head for head in lists if head is not None]
        heapq.heapify(least_heap)

        dummy_head = ListNode()
        cur = dummy_head    # 用于追踪结果链表当前的末尾，用于添加下一个节点

        while least_heap:   # 遍历直到堆空
            node = heappop(least_heap)  # 下一个最小的节点
            # 将其在原本链表中的下一个节点（如有）存入堆中，作为新的下一个最小节点的候选者
            if node.next:
                heappush(least_heap, node.next)
            # 将当前的最小节点node加入结果链表中
            cur.next = node
            cur = cur.next

        return dummy_head.next
```

Eg4. 会议室II

给你一个会议时间安排数组`intervals`，每个会议时间都包括开始和结束的时间`intervals[i] = [start_i, end_i]`，返回所需会议室的最小数量。

示例：

```
输入：intervals = [[0, 30], [5, 10], [15, 20]]
输出：2
```

解：

首先根据会议开始时间从早到晚对各个子区间做排序。然后，设置一个最小堆`free_rooms`，其元素数量表示当前占用的会议室数量，初始化时输入第0个区间的结束时间，然后从第1个区间开始往后遍历：若当前区间的起始时间晚于（或等于）堆顶，那么说明此时堆顶对应的那个区间的结束时间早于当前区间的开始时间，因此当前区间可以复用堆顶对应区间的会议室，因此可以结束堆顶区间的会议（弹出堆顶，表示释放一个会议室），然后将当前区间的结束时间入堆（不管有没有弹出堆顶，表示当前新增一个会议室）。最终，堆中元素数量即为需要的会议室数量。

```python
import heapq

def minMeetingRooms(intervals):
    # 如果会议安排列表为空，直接返回0
    if not intervals:
        return 0

    # 初始化一个空的最小堆
    free_rooms = []

    # 先根据会议的开始时间对会议进行排序
    intervals.sort(key=lambda x: x[0])

    # 将第一个会议的结束时间加入到最小堆中
    # 这表示目前我们有一个会议室被占用，直到这个时间点
    heapq.heappush(free_rooms, intervals[0][1])

    # 从第二个会议开始遍历
    for i in intervals[1:]:

        # 如果当前会议的开始时间大于等于最小堆中的最早结束时间
        # 说明这个会议室可以被重复使用
        # 因此我们可以移除堆顶元素（最早结束的会议室）
        if i[0] >= free_rooms[0]:
            heapq.heappop(free_rooms)

        # 将当前会议的结束时间加入最小堆
        # 表示新增一个会议室，或是延续使用原会议室
        heapq.heappush(free_rooms, i[1])

    # 堆中元素的数量，就是我们需要的会议室数量
```

```
        return len(free_rooms)
```

桶

Eg1. 最大间距

给定一个无序的数组 `nums`，返回 *数组在排序之后，相邻元素之间最大的差值* 。如果数组元素个数小于 2，则返回 `0` 。

您必须编写一个在「线性时间」内运行并使用「线性额外空间」的算法。

示例：

```
输入: nums = [3,6,9,1]
输出: 3
解释: 排序后的数组是 [1,3,6,9], 其中相邻元素 (3,6) 和 (6,9) 之间都存在最大差值 3。
```

解：

如果先排序（例如使用快排）再依次计算相邻元素之间的差值的话，排序这步就已经达到$O(N \log N)$，不符合题目的线性复杂度要求。

因此，可以考虑类似桶排序的思路：首先用$O(N)$复杂度分桶，然后只要确保最大间隔一定出现在桶间即可，这样每个桶只需要维护其中的最大元素和最小元素即可，这样只需再依次求出各个桶的最大值和下个桶的最小值之间的差，即可筛选出最大间隔。

为了保证最大间隔一定出现在桶间，需要设置桶的大小为：

```
bucket_size = max(1, (max_val-min_val) // (n-1))
```

这是因为，根据鸽巢原理，`n`个元素存在`n-1`个间隔，在这些元素完全均匀分布的情况下，最大间隔取最小值，也即将数据范围平均到每个间隔上的平均值：`(max_val-min_val)/(n-1)`。因此，只要确保桶的大小小于等于`(max_val-min_val)/(n-1)`，就能确保最大间隔一定不出现在桶内，而是出现在某两个桶之间。

```python
class Solution:
    def maximumGap(self, nums: List[int]) -> int:
        n = len(nums)
        if n < 2:
            return 0

        # 首先，统计最小值和最大值来确定数据范围，然后即可求出桶大小以及桶数量
        min_val = min(nums)
        max_val = max(nums)

        if min_val == max_val:
            return 0

        bucket_size = max(1, (max_val - min_val) // (n - 1))
```

```
        bucket_num = (max_val - min_val) // bucket_size + 1

        # 构建桶，每个桶维护两个值：桶中最小值和桶中最大值，第i个桶的最小值和最大值分别为buckets[i][0]和buckets[i][1]
        buckets = [[None, None] for _ in range(bucket_num)]

        # 然后，依次遍历每个元素，并将其放到其应该在的桶中
        for num in nums:
            idx = (num - min_val) // bucket_size
            # 将该元素入桶，更新桶的最大值和最小值
            if buckets[idx][0] is None:
                buckets[idx][0] = num
                buckets[idx][1] = num
            else:
                if num < buckets[idx][0]:
                    buckets[idx][0] = num
                if num > buckets[idx][1]:
                    buckets[idx][1] = num

        # 然后即可依次遍历各桶，并统计每个桶的最大值和下一个桶最小值之间的间隔，并更新当前统计到的最大间隔值
        max_gap = 0
        prev_max = buckets[0][1]
        for i in range(1, bucket_num):
            if buckets[i][0] is None:
                continue
            current_min = buckets[i][0]
            gap = current_min - prev_max
            if gap > max_gap:
                max_gap = gap
            prev_max = buckets[i][1]

        return max_gap
```

分治

基础：快速排序与归并排序

Eg 1 数组中的第K个最大元素

给定整数数组 `nums` 和整数 `k`，请返回数组中第 `k` 个最大的元素。

请注意，你需要找的是数组排序后的第 `k` 个最大的元素，而不是第 `k` 个不同的元素。

你必须设计并实现时间复杂度为 `O(n)` 的算法解决此问题。

解：（数组中前/第k大/小元素通用模板）

借鉴快速排序的思路：可以进行若干次分割，每次分割都确保分割点左侧的元素都小于等于分割点，分割点右侧的元素都大于等于分割点，而左右区间内部的顺序则无所谓。相比于快速排序需要分割到底来完成完整的排序，本问题则只需要找到分割点`i==k`时，即可确保`k`位置左侧的`k`个元素都是小于当前元素的，这样即说明`nums[i]`就是数组中第k个最小元素，可以返回并不需要进行进一步分割操作。保持partition部分不变，将其稍加改动即可适用于数组中前/第k大/小元素问题。

本题找第k个最大元素：

```
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # 分割部分和快排完全一样
        def partition(nums, left, right):
            i, j = left, right
            while i < j:
                while i < j and nums[j] >= nums[left]:
                    j -= 1
                while i < j and nums[i] <= nums[left]:
                    i += 1
                nums[i], nums[j] = nums[j], nums[i]
            nums[i], nums[left] = nums[left], nums[i]
            return i

        def topk_split(nums, k, left, right):
            if left >= right:
                return
            i = partition(nums, left, right)
            # 若i==k，则分割完成，此时k左侧的0~k-1这k个元素都小于k
            if i == k:
                return
            # 若k处于右半段，则进一步对右半段进行分割
            if i < k:
                topk_split(nums, k, i+1, right)
            # 若k处于左半段，则进一步对左半段进行分割
            elif i > k:
                topk_split(nums, k, left, i-1)

        # 相当于找第len(nums)-k大的元素
        k = len(nums) - k
        topk_split(nums, k, 0, len(nums)-1)
        return nums[k]
```

二分查找

二分查找的本质是根据序列本身的某些规律（如单调递增等），使得每次二分后都有条件能判定目标处于左右哪一段中，从而缩小搜索区间。只要数组的规律能够使得存在这样的条件，那么就可以二分查找。

Eg1 搜索插入位置

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

请必须使用时间复杂度为 `O(log n)` 的算法。

解：题目要求即为：找到第一个大于等于目标值的数组元素的index并返回（不管是在数组中确切找到目标值，还是目标值原本不存在而需要将其插入数组中）。

使用二分法：定义`left`和`right`指针，当`left<=right`时，找到其中点`mid`，然后判断`nums[mid]`和`target`的大小关系：

- 若`nums[mid] < target`，则说明目标位于`mid+1~right`之间，因此将`left`设为`mid+1`从而缩小区间
- 若`nums[mid] >= target`，则说明目标位于`left~mid`之间，此时将right设为`mid-1`。

  对于边界情况：

  - 如果此时`nums[mid] == target`，那么`right=mid-1`后，`left`必然一直向右移动，直到由`left==right==mid`变到`left+1 > right`，则此时循环退出，`left`就是`target`的index
  - 如果此时`nums[mid] > target`，且`nums[mid-1] = nums[right] < target`，则`left`之后也必然会一直向右移动，直到由`left==right==mid`变到`left+1 > right`，则此时循环退出，`left`就是第一个比`target`大的元素的index

因此，循环退出（`left>right`）后，无论是什么情况，返回`left`即为所求结果。

```
class Solution(object):
    def searchInsert(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left = 0
        right = len(nums) - 1
        # 想要找到第一个大于等于target的位置
        while left <= right:
            # 中点的index
            mid = (left + right) // 2

            if nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1

        return left
```

一种更好理解的写法：

```
class Solution(object):
    def searchInsert(self, nums, target):
        """
```

```
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left = 0
        right = len(nums) - 1
        # 想要找到第一个大于等于target的位置
        while left <= right:
            # 中点的index
            mid = (left + right) // 2
            # 若中点小于目标，则说明目标位于mid+1~right之间，因此将left设为mid+1从而缩小区间
            if nums[mid] < target:
                left = mid + 1
            # 若中点大于目标，则说明目标位于left~mid-1之间，因此将right设为mid-1从而缩小区间
            elif nums[mid] > target:
                right = mid - 1
            # 若中点等于目标则恰好命中，返回mid作为index即可
            else:
                return mid

        # 最后跳出循环的条件是left>right，此时必有left=right+1，也即第一个大于target的元素的index
        return left
```

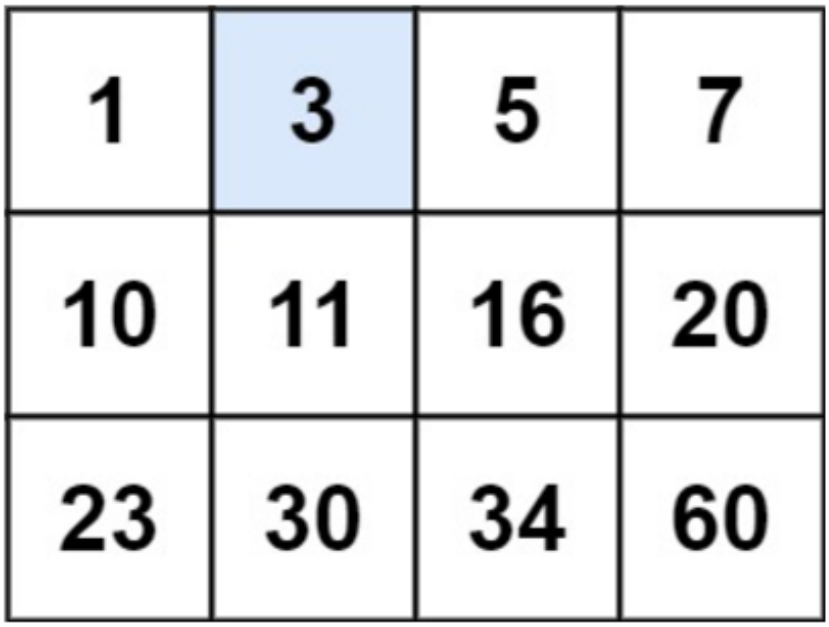
总之，需要考虑的是最后二分到极限的时候，循环的退出条件以及退出时的情形是什么，应该返回`left`还是`right`。

Eg2. 搜索二维矩阵

给你一个满足下述两条属性的 `m x n` 整数矩阵：

- 每行中的整数从左到右按非严格递增顺序排列。
- 每行的第一个整数大于前一行的最后一个整数。

给你一个整数 `target` ，如果 `target` 在矩阵中，返回 `true` ；否则，返回 `false` 。



> **输入:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
> **输出:** true

解：本题为二分查找的二维情况，只需进行两遍二分查找即可：

第一遍对于第一列元素进行查找，用来定位目标可能处于的行数。由于第一列本身一定是升序排列，因此只需对其进行二分查找，找到第一个小于等于目标的元素即可（可以直接套用上题的模板，找到是第一个大于等于目标的元素，将其-1即为第一个小于目标的元素。当然取等情况需要额外判断），这样说明目标只可能在该元素对应的行中。

第二遍对于上述行再进行一次二分查找，看有无恰好等于目标的元素。

```
class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        # 查找第一列中第一个小于等于该元素的位置
        m, n = len(matrix), len(matrix[0])
        first_col = [matrix[i][0] for i in range(m)]
        left = 0
        right = m - 1
        while left <= right:
            mid = (left + right) // 2
            if first_col[mid] < target:
                left = mid + 1
            elif first_col[mid] > target:
                right = mid - 1
            else:
                # 已找到target，直接返回True
                first_ele_idx = mid
```

```
                return True
        first_ele_idx = left - 1

        # 在该行中查找是否存在该元素
        target_row = matrix[first_ele_idx]
        left = 0
        right = n - 1
        while left <= right:
            mid = (left + right) // 2
            if target_row[mid] < target:
                left = mid + 1
            elif target_row[mid] > target:
                right = mid - 1
            else:
                first_ele_idx = mid
                return True
        return False
```

Eg 2.2 搜索二维矩阵II

编写一个高效的算法来搜索 `m x n` 矩阵 `matrix` 中的一个目标值 `target` 。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

也即，相比于“搜索二维矩阵”来说，不保证上一行的末尾一定比下一行的开头小，只能确保每行内部具有升序关系

若该目标值在矩阵中则返回True，反之返回False。

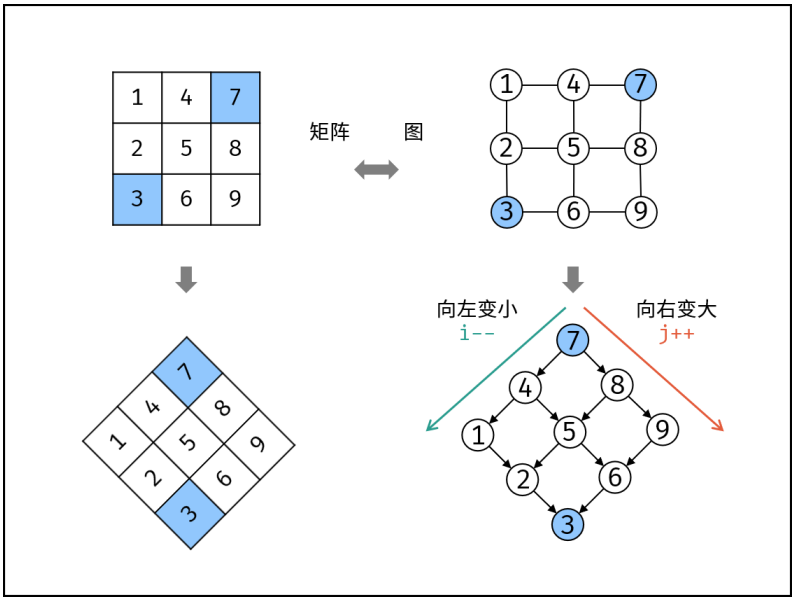
示例：

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

解：

本题没法像上一题那样只需对列和行分别做一轮二分就能完成，因为本题的各行拼起来后并不能成为一个单调递增的1d序列。因此，可以对于每一行分别做二分来搜索，不过这种方法比较笨。

本题还有一种更巧妙的解法：将矩阵逆时针旋转45度，然后将每个矩阵元视为节点，可以发现其构成的结构满足二叉搜索树的定义：任何节点的左子节点都小于其本身，右子节点都大于其本身：



因此，可以将其视为二叉搜索树进行贪心搜索：从根节点（也即原矩阵的右上角元素）开始，设当前节点在原矩阵中的行列数分别为`i,j`，则：若当前节点小于target，则往“二叉搜索树的右下方”移动1位，对应原矩阵中也即行数`i+=1`；若当前节点大于target，则往“二叉搜索树的左下方”移动1位，对应原矩阵中也即列数`j-=1`；若当前节点等于target，则说明搜到了，返回`True`即可。遍历过程中，如果发现某个时刻`i`或`j`超出了矩阵边界范围，则说明该矩阵中不存在target，返回`False`

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        i, j = 0, len(matrix[0]) - 1    # 初始化为矩阵右上角的位置（也即“二叉搜索树的root节点”）

        while i < len(matrix) and j >= 0:
            if matrix[i][j] < target:
                # 若当前值小于目标值，则向“右下方”走
                i += 1
            elif matrix[i][j] > target:
                # 若当前值大于目标值，则向“左下方”走
                j -= 1
            else:   # 若当前值等于目标值，则说明搜到了
                return True

        # 若i,j中的某一个越界了还没找到，那就说明target不在矩阵中
        return False
```

Eg 3 搜索旋转排序数组

整数数组 `nums` 按升序排列，数组中的值 **互不相同** 。

在传递给函数之前，`nums` 在预先未知的某个下标 `k`（`0 <= k < nums.length`）上进行了 **旋转**，使数组变为 `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]`（下标 **从 0 开始** 计数）。例如，`[0,1,2,4,5,6,7]` 在下标 `3` 处经旋转后可能变为 `[4,5,6,7,0,1,2]` 。

给你 **旋转后** 的数组 `nums` 和一个整数 `target` ，如果 `nums` 中存在这个目标值 `target` ，则返回它的下标，否则返回 `-1` 。

你必须设计一个时间复杂度为 `O(log n)` 的算法解决此问题。

解：只要能够找到判定目标处于左右哪一段的条件，就能够使用二分法来搜索。观察`nums`序列，发现虽然其整体上不是单调递增的，但在任何分割下，左右两段中必然至少有一段是单增的，因此可以先通过比较其中某一段的首尾元素大小来确定哪一段是单增的，然后即可进一步判定target所处区间：

- 如果`nums[mid] == target`：搜索到了结果，返回`mid`即可
- 如果`nums[left] <= nums[mid]`：左半段是递增的
  - 如果`nums[left] <= target < nums[mid]`：说明target在左半段，因此可以通过设置`right = mid - 1`来将搜索区间缩小到左半段
  - 否则说明target在右半段，因此可以通过设置`left = mid + 1`来将搜索区间缩小到右半段
- 相对地，若右半段有序，则可以以类似的方法进行区间判断与缩小。

```
class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            # 若nums[mid]就是要查找的元素，则说明搜索到了，直接返回即可
            if nums[mid] == target:
                return mid
            # 下面需要确认哪一段是有序的，然后即可判定target位于哪一段，从而缩小搜索区间
            # 若左半段是有序的
            if nums[left] <= nums[mid]:
                # 确认target是否在这有序的左段中
                # 若target在有序的左段中
                if target >= nums[left] and target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            # 若右半段是有序的
            else:
                # 若target在有序的右段中
                if target > nums[mid] and target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1
        return -1
```

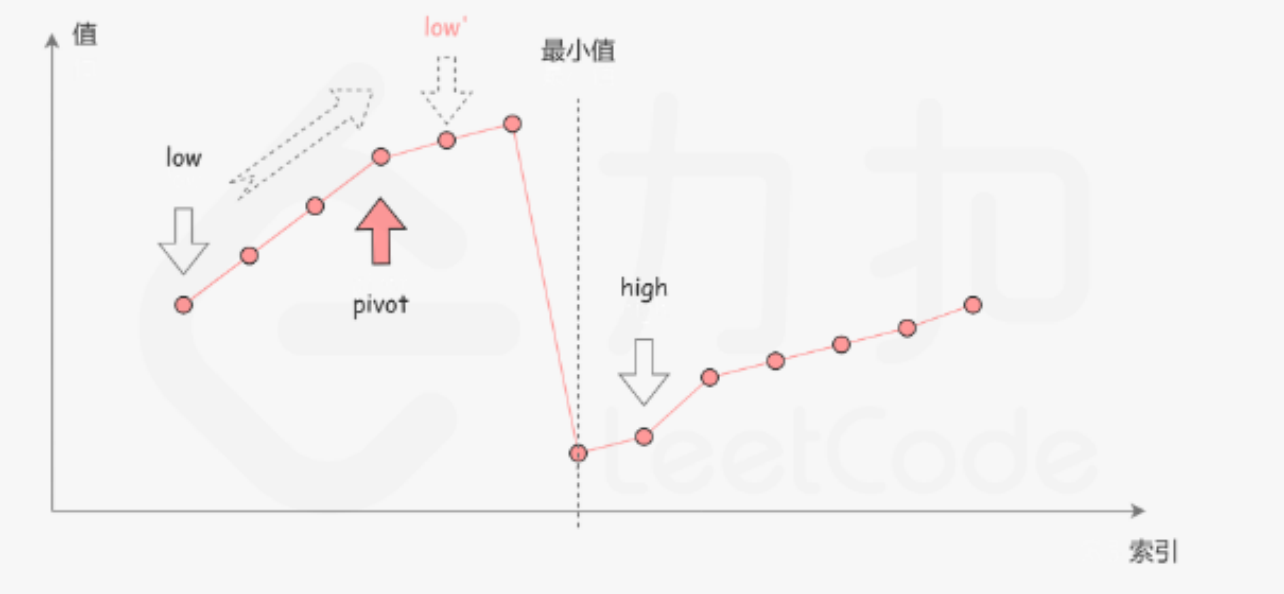
Eg 4. 寻找旋转排序数组中的最小值

对于在某个位置旋转后的有序数组（和上题定义一样，元素各不相同，如`[4,5,6,7,0,1,2]`，也可能不旋转），返回其最小值。

解：方便起见，维护一个最小值变量进行记录。

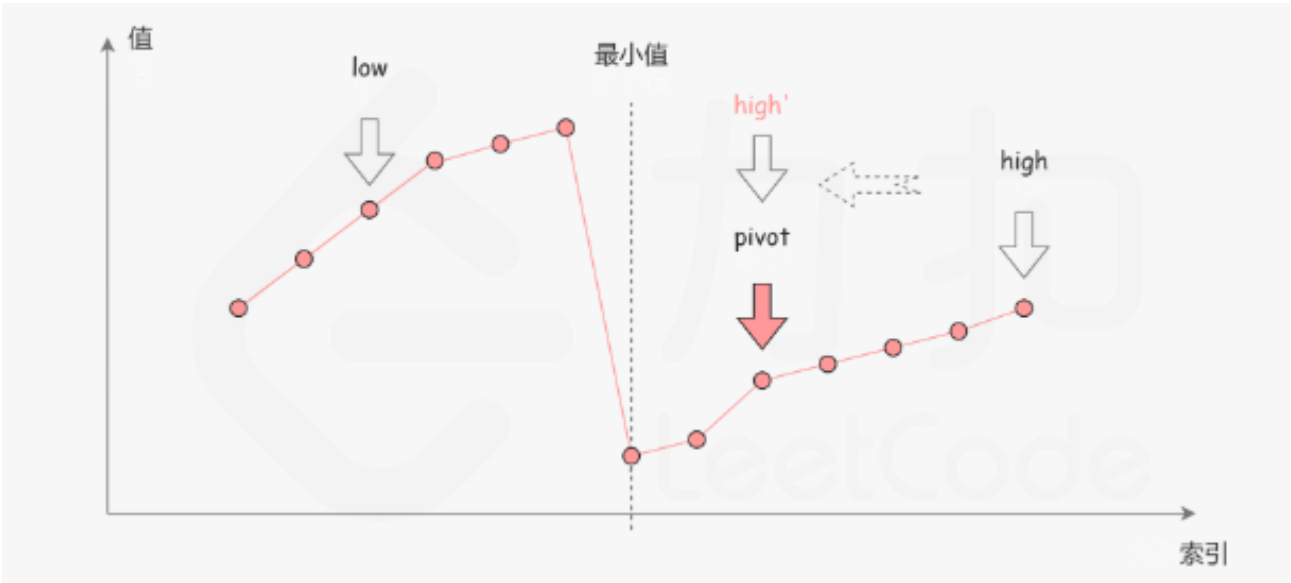
观察序列可发现，使用mid将序列分成两段后，则：

- 若`nums[mid] > nums[right]`，则最小值一定在右半段。且`nums[mid]`必然不是最小值，因此这里只缩小区间而不需要更新最小值变量



- 若`nums[mid] <= nums[right]`，则说明右半段递增，最小值一定在左半段，或就是`nums[mid]`本身。因此一方面缩小区间到mid左侧，另一方面更新最小值变量的记录。

  （注：之所以最后不能返回循环退出时的`nums[mid]`作为结果，是因为如果某一轮的`nums[mid]`本身就是最小值的话，那么`right=mid-1`后就会错过这个最小值。因此，设置一个最小值变量来迭代记录是最保险的）




```
class Solution(object):
    def findMin(self, nums):
```

```
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums) - 1
        min_val = 10 ** 6

        while left <= right:
            mid = (left + right) // 2
            # 若nums[mid] > nums[right]，则最小值（低谷）一定在右半段
            if nums[mid] > nums[right]:
                left = mid + 1
            # 若nums[mid] <= nums[right]，则一定在左半段或就是mid本身
            else:
                min_val = min(min_val, nums[mid])
                right = mid - 1

        return min_val
```

Eg 5. 在排序数组中查找元素的第一个和最后一个位置

给你一个按照非递减顺序排列的整数数组 `nums`，和一个目标值 `target`。请你找出给定目标值在数组中的开始位置和结束位置。

如果数组中不存在目标值 `target`，返回 `[-1, -1]`。

你必须设计并实现时间复杂度为 `O(log n)` 的算法解决此问题。

例如：

```
输入：nums = [5,7,7,8,8,10], target = 8
输出：[3,4]
```

解：也即想要找到target第一次出现和最后一次出现的位置。

以找到第一次出现的位置为例，首先使用二分法先找到一个target位置（也即`nums[mid] == target`），则说明：target第一次出现的位置要么就是这个位置本身（`mid`），要么就比这个位置靠左，则只需先把第一次出现位置临时设为该位置（`first == mid`），然后进一步取左侧区间（`right = mid - 1`）再搜索一下看看还有没有更靠前的`target`出现，从而迭代更新`first`。

找到最后一次出现的位置思路类似。

```
class Solution(object):
    def searchRange(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
```

```
        if not nums: return [-1, -1]
        first, last = -1, -1

        # 寻找第一次出现target
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            # 如果mid恰好就是target，则说明第一个target只能是mid自己或者在mid左侧
            # 因此先将first赋值为mid，然后进一步缩小搜索区间为mid左侧，看看能不能找到更早出现的
            if nums[mid] == target:
                first = mid
                right = mid - 1
            # 若nums[mid] > target，则说明第一个target必然在mid左侧
            elif nums[mid] > target:
                right = mid - 1
            # 若nums[mid] < target，则说明第一个target必然在mid右侧
            else:
                left = mid + 1

        # 寻找最后一次出现target
        left, right = 0, len(nums) - 1
        while left <= right:
            mid = (left + right) // 2
            # 如果mid恰好就是target，则说明最后一个target只能是mid自己或者在mid右侧
            # 因此先将last赋值为mid，然后进一步缩小搜索区间为mid右侧，看看能不能找到更晚出现的
            if nums[mid] == target:
                last = mid
                left = mid + 1
            # 若nums[mid] > target，则说明最后一个target必然在mid左侧
            elif nums[mid] > target:
                right = mid - 1
            # 若nums[mid] < target，则说明第一个target必然在mid右侧
            else:
                left = mid + 1

        return first, last
```

Eg 6. 寻找峰值

峰值元素是指其值严格大于左右相邻值的元素。

给你一个整数数组 `nums`，对于所有有效的 `i` 都有 `nums[i] != nums[i + 1]`，找到峰值元素并返回其索引。数组可能包含多个峰值，在这种情况下，返回 **任何一个峰值** 所在位置即可。

你可以假设 `nums[-1] = nums[n] = -∞` 。

你必须实现时间复杂度为 `O(log n)` 的算法来解决此问题。

解：

由于相邻元素都不相同，因此必然存在峰值。峰值存在的判别式即为：`nums[i-1]< nums[i] > nums[i+1]`，只需找到一个这样的峰值即可。

进一步，观察到：若`nums[i] < nums[i+1]`则说明`nums[i]`处于上坡，则`[i+1, len(nums))`范围内必然存在至少一个峰值；若`nums[i] < nums[i-1]`则说明`nums[i]`处于下坡，则`[0, i-1]`范围内必然存在至少一个峰值。由此即可用二分法来不断缩小搜索区间，直到找到符合条件的峰值。

```
class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """

        # 辅助函数，输入下标 i，返回 nums[i] 的值
        # 方便处理 nums[-1] 以及 nums[n] 的边界情况
        def get(i):
            if i == -1 or i == len(nums):
                return float('-inf')
            return nums[i]

        left, right = 0, len(nums)-1
        while left <= right:
            mid = (left + right) // 2
            # 若mid本身就是峰值位置，则直接返回即可
            if get(mid) > get(mid - 1) and get(mid) > get(mid + 1):
                return mid
            # 若是上坡，则[mid+1, len(nums))中一定存在峰值，可将范围缩小到mid+1及以后
            if get(mid) < get(mid + 1):
                left = mid + 1
            # 若是下坡，则[0, mid)中一定存在峰值，可将范围缩小到mid-1及以前
            elif get(mid) < get(mid - 1):
                right = mid - 1
```

Eg7. 寻找重复数

给定一个包含 `n + 1` 个整数的数组 `nums` ，其数字都在 `[1, n]` 范围内（包括 `1` 和 `n`），可知至少存在一个重复的整数。

假设 `nums` 只有 **一个重复的整数** ，返回 **这个重复的数** 。

你设计的解决方案必须 **不修改** 数组 `nums` 且只用常量级 `O(1)` 的额外空间。

解：

由于要求O(1)空间，因此不能使用set或哈希表等来具体记录和统计`[1,n]`中每个元素在`nums`中是否出现/出现次数，只能使用类似`count`计数的方式来统计某个指标。

因此，本题可以对取值区间数组`[1,2,...,n]`进行二分查找（不是对原数组`nums`进行二分，原数组本身没有任何规律）。具体而言，初始化区间端点`left, right = 1, n`。在每个取值区间`[left, right]`内，找到其中点值`mid`后，遍历一遍`nums`数组并统计有多少个数字小于等于`mid`；遍历完成后，设有`count`个数字小于等于`mid`，若`count <= mid`，则说明`[left, mid]`中没有重复的数字（`left,...,mid`每个数字最多出现了一次），说明重

复的数字处于`[mid+1, right]`中；若`count>mid`，则说明重复的数字就处于`[left, mid]`中。

（有点没太懂）

```
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        n = len(nums) - 1
        left, right = 1, n  # 初始搜索区间是[1, n]

        # 开始二分搜索
        while left < right:
            mid = (left + right) // 2   # 当前搜索区间中点
            count = 0   # 统计原数组中，有多少数字是小于等于区间中点值的
            for num in nums:
                if num <= mid:
                    count += 1

            # 如果数组中小于等于区间中点值的数字数量count<=mid，则说明数组中小于等于区间中点值的数字left,...,mid中没有重复的，重复数字在[mid+1, right]中
            if count <= mid:
                left = mid + 1
            # 如果数组中小于等于区间中点值的数字数量count>mid，则说明重复值就在[left, mid]之间
            else:
                right = mid
        return left
```

Eg 8. 寻找两个正序数组的中位数

给定两个大小分别为 `m` 和 `n` 的正序（从小到大）数组 `nums1` 和 `nums2`。请你找出并返回这两个正序数组的 **中位数** 。

算法的时间复杂度应该为 `O(log (m+n))` 。

示例：

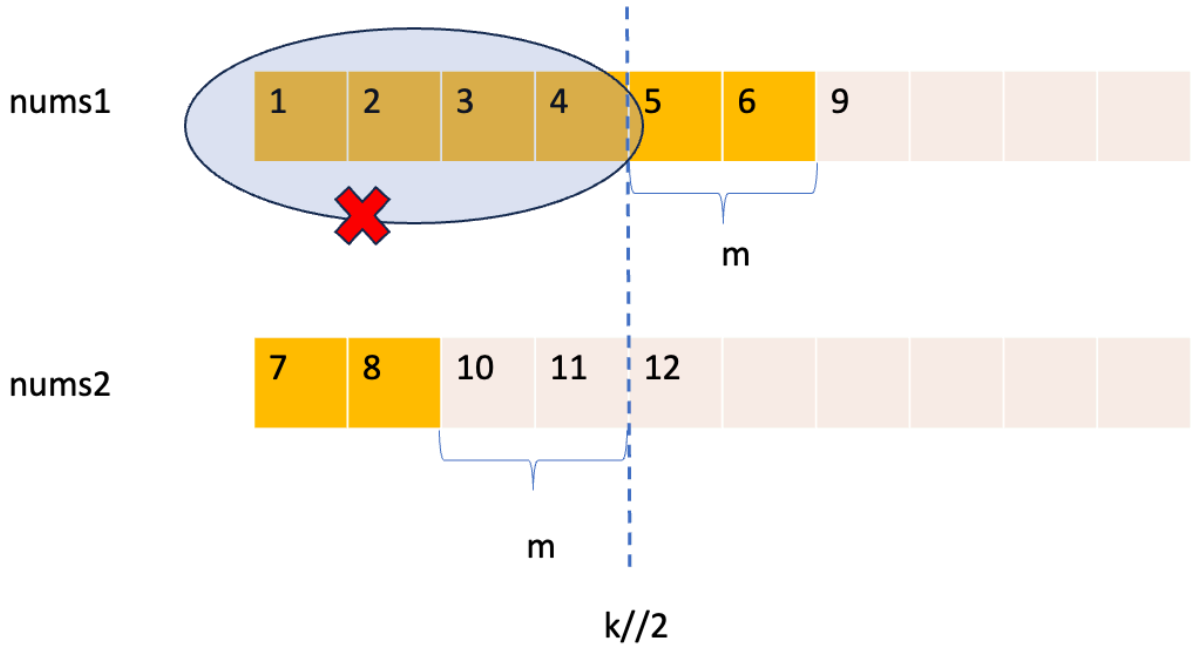
```
输入：nums1 = [1,2], nums2 = [3,4]
输出：2.50000
解释：合并数组 = [1,2,3,4] ，中位数 (2 + 3) / 2 = 2.5
```

解：

当两个数组的长度之和为奇数时，中位数即为全局（两个数组中）第`k = (len(nums1) + len(nums2)) // 2 + 1`小的数；当两个数组长度之和为偶数时同理。此时，问题转化成了在全局寻找第`k`小的数。

下面考虑如何在给定的两个有序数组`nums1, nums2`中找到第k小的元素（以下提到的`k`都是从1开始算的）。具体而言，在两个数组中分别设置一个指针`i,j`，它们初始化时都位于最左边（`i,j=0,0`），这两个指针的右侧分别为当前在两个数组中剩余的搜索空间，接下来就要想办法通过某种标准来右移这两个指针，来不断缩小剩余的搜索空间。

一开始时，希望在`i,j=0,0`右侧的区域寻找第k小的数，假设事实上前k小的数字在`nums1`中占`k//2+m`个，在`nums2`中占`k//2-m`个，那么此时体现为`nums1[k//2] <= nums2[k//2]`，且从示意图中可以看出当前搜索区域中第k小的数字一定不是位于`nums1[i: i+k//2]`这部分，因此可以将这部分从搜索空间中排除，体现为将指针`i`右移为`i = i+k//2+1`，也即`nums1`中剩余的搜索部分缩小了。搜索空间缩小后，由于排除了`k//2`个元素（且这些元素是小于k的），因此问题转换为在剩余的搜索空间中找到第`k_now = k - k//2`小的元素。后边即类似地重复这个迭代过程，不断进行比较和缩小搜索区间（并相应地更新在剩余搜索区间中需要找到第几小的元素`k_now`），直到：某一个数组中的搜索区间全部被排除完，或`k_now`缩小为`k_now=1`，此时可以直接找到最终答案。



因此，每轮迭代的算法流程为：

- 给定当前两个数组的搜索区间左端点`i,j`，以及当前需要在总搜索区间中找到第`k_now`小的数
- 首先检查当前是否到达了边界条件：
  - 边界条件1：如果发现`i==len(nums1)`或`j==len(nums2)`，也即`nums1`或`nums2`已经彻底被排除完了，当前总的搜索区间就等于`nums2`或`nums1`剩余的搜索区间，因此只需返回`nums2`或`nums1`剩余搜索区间中的第`k_now`个元素即可，也即`nums2[j+k_now-1]`或`nums1[i+k_now-1]`
  - 边界条件2：如果发现虽然当前两个数组都没有被完全排除掉，但`k_now`缩小到了`k_now==1`，那么说明只需在剩余的总搜索区间中找到最小的元素即可，由于两个数组的剩余搜索区间各自都是单调递增的，因此最小元素只能出自两个剩余搜索区间的第1个元素，在二者中取最小的返回即可：`min(nums1[i], nums2[j])`
- 如果没有到达边界条件，则去比较两个数组剩余搜索区间中，各自的左端点偏移`k_now//2`的位置（为了防止偏移后超过右边界，因此这个偏移后的位置要和右边界取min）：

```
i_tmp = min(i + (k_now // 2) - 1, len(nums1)-1)
j_tmp = min(j + (k_now // 2) - 1, len(nums2)-1)
```

  对于较小的一方，则将其偏移点左边的部分从搜索空间中移除（`i=i_tmp+1`或`j=j_tmp+1`），并更新`k_now`，将其减去这次移除的元素数量

```python
class Solution:
    def findKthElementInTwoArrays(self, nums1, nums2, k):
        """
        该函数用于寻找nums1和nums2中，总的（或者说二者合并后的）第k个最小元素
        """
        i, j = 0, 0 # i, j分别表示此时nums1, nums2中的指针，它们指针右侧的部分是目前剩下的搜索区域
        k_now = k   # k_now表示当前需要在剩下的搜索区域中找到第k_now小的数，其就是总的nums1和nums2中的第k小的数
        while True:
            # 首先检测此时某个数组是否到达了边界，或k_now到达了1，此时可以返回结果
            if i == len(nums1):     # 如果nums1已经被排除完了，则总的剩余搜索区域就是nums2的剩余区域了，只需要找到nums2剩余区域中第k_now小的数（从0开始算的话也即第k_now-1个元素）即可得到最终答案
                return nums2[j + k_now - 1]
            if j == len(nums2):   # 同理，如果nums2已经被排除完了，则找出nums1剩余区域的第k_now-1个元素即可
                return nums1[i + k_now - 1]
            if k_now == 1:        # 或者如果此时k_now==1，则说明只需找到nums1和nums2总剩余区域中的最小的数即可，它就是两个剩余区域各自的首个元素中的最小值
                return min(nums1[i], nums2[j])

            # 然后考虑正常情况
            # 每次比较两个数组剩余搜索空间中第 k_now//2 小的元素（如果存在），较小的那个数组的前 k_now//2 个元素不可能是第 k_now 小的元素，因此可以排除这部分，然后在剩下的部分中继续查找第 k_now - len(排除掉的元素数量) 小的元素（也即把k_now更新为这个新值）
            i_tmp = min(i + (k_now // 2) - 1, len(nums1)-1)
            j_tmp = min(j + (k_now // 2) - 1, len(nums2)-1)
            if nums1[i_tmp] <= nums2[j_tmp]:    # 如果是nums1中的部分被排除
                # 排除掉nums1[i: i_tmp] 这部分，更新i，并将k_now也进行更新
                k_now = k_now - (i_tmp - i + 1)
                i = i_tmp + 1
            else:   # 如果是nums2中的部分被排除
                # 排除掉nums2[j: j_tmp] 这部分，更新j，并将k_now也进行更新
                k_now = k_now - (j_tmp - j + 1)
                j = j_tmp + 1


    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        if (len(nums1) + len(nums2)) % 2 == 1:  # 如果两数组的总长度为奇数
            k = (len(nums1) + len(nums2)) // 2 + 1      # 中位数就是nums1和nums2中第k小的数（从1开始算）
            median = self.findKthElementInTwoArrays(nums1, nums2, k)
        else:       # 如果两数组的总长度为偶数
            k1 = (len(nums1) + len(nums2)) // 2
            k2 = (len(nums1) + len(nums2)) // 2 + 1
            median = (self.findKthElementInTwoArrays(nums1, nums2, k1) + self.findKthElementInTwoArrays(nums1, nums2, k2)) / 2
        return median
```

回溯

回溯法通常用于解决组合问题。其核心思想是构造解空间树，逐个尝试每个可能的选择，在每一步递归后退回上一步（撤销当前选择），再继续进行其他选择尝试。

回溯法是一种**试探-撤销-继续尝试**的递归过程。其框架包括以下几个关键部分：

1. **选择路径**：遍历将每一个下一步可能加入组合的元素，逐个尝试将其加入当前组合。
2. **递归探索**：每次加入一个数字后，递归调用处理剩余数字。
3. **回溯撤销**：当递归返回时，撤销选择，继续探索其他可能（继续遍历下一个可能加入组合的元素）。

Eg 1. 组合

给定两个整数 `n` 和 `k`，返回范围 `[1, n]` 中所有可能的 `k` 个数的组合。

你可以按 **任何顺序** 返回答案。

示例 1：

```
输入：n = 4, k = 2
输出：
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

解：

主函数中：

- 定义`res = []`，用于储存所有的组合

  如：`res = [[2,4], [3,4], ...]`
- 定义`combination = []`，用于存储当前正在形成中的（可能还不完整的）组合

  如：`combination = [2,4]`、`combination = [2]`
- 调用递归辅助函数`backtrace`，从0开始构造组合：`self.backtrace(0, n, k, combination, res)`

```python
    def combine(self, n, k):
        res = []  # 用于存储所有结果
        combination = []  # 当前的组合
        self.backtrace(0, n, k, combination, res)
        return res
```

递归辅助函数定义如下：

```
    def backtrace(self, i, n, k, combination, res):
        # 如果当前组合的大小等于 k，则说明构造完成了一个组合，将组合加入结果集
        if len(combination) == k:
            res.append(combination[:])  # 注意要传入组合的副本
            return

        # 从 i + 1 开始尝试下一步所有可能的数字
        for j in range(i + 1, n + 1):
            combination.append(j)  # 选择当前数字
            self.backtrace(j, n, k, combination, res)  # 递归
            combination.pop()  # 撤销选择（回溯）
```

其中：

- 终止条件：如果当前正在形成中的组合`combination`长度到达`k`，说明该组合已经构造完全，可以将其加入`res`中。

  > 注意由于`combination`会被一直复用，接下来还会进行回溯等操作，因此为了不影响`res`中已保存好的结果，应该将其副本`combination[:]`而不是本身添加到`res`中）

- 如果当前组合的长度还没到达`k`，则从当前尝试区间的左端点的下一位`j=i+1`开始，对于组合的下一位进行遍历尝试。每尝试一个`j`，则先将其加入当前组合`combination`，然后再以`j`为左端点进一步进行递归回溯。解决完`j`后边的所有组合情况之后，将`j`撤回，再尝试将`j+1`填入当前的组合位置进行下一轮尝试，如此往复。

  ```
  for j in range(i + 1, n + 1):
      combination.append(j)
      self.backtrace(j, n, k, combination, res)  # 递归
      combination.pop()  # 撤销选择（回溯）
  ```

Eg2. 子集

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例：

```
输入：nums = [1,2,3]
输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

解：

本题和“组合”本质上是一样的，只不过没有了组合大小的限制，所有组合不管长度大小都是独特的，那么就都是子集，因此可以无条件将每个构造出来的组合都收入最终结果中。

```
class Solution:
    def backtrace(self, i, combination, nums, results):
```

```
        # 每构造出的一个组合都是独特的，因此都将其收入最终的results中
        results.append(combination[:])

        for j in range(i, len(nums)):
            combination.append(nums[j])
            self.backtrace(j+1, combination, nums, results)
            combination.pop()


    def subsets(self, nums: List[int]) -> List[List[int]]:
        results = []
        combination = []
        self.backtrace(0, combination, nums, results)
        return results
```

Eg 3 全排列

给定一个不含重复数字的数组`nums`，返回其所有可能的全排列 。你可以 按任意顺序 返回答案。

解：仍使用回溯法。其与上述组合问题的主要区别在于：组合问题中可以通过移动`i,j`等来依次往后尝试，从而避免重复。而对于全排列问题，其每一个排列都会包括`nums`中的所有元素，也即当一个全排列形成到某个位置时，其下一个要添加的元素既有可能在当前位置之前也有可能在当前位置之后——只要不和当前全排列中已有元素重合即可。

因此，这里额外维护一个`used`列表，其用于记录每个`nums[i]`在当前全排列中是否已经被用过，从而知道哪些元素可以被继续加入当前组合。这样一来，在回溯函数中，循环尝试该全排列的下一位时，每次都可以从头到尾遍历`nums`中的所有元素，只需在遍历到一个元素时使用`used`列表检查一下它是否已经被用过即可：如果没用过的话，那么可以尝试将其作为下一位加入当前全排列，并且本轮中需要将其标记为用过（`used[j] = True`），然后再对当前全排列继续递归，最后再将当前元素撤回，同时也将其从`used`中删除（`used[j] = False`）

```
class Solution:
    def backtrace(self, nums, used, permutation, res):
        # 如果当前排列的大小等于 nums 的大小，保存结果
        if len(permutation) == len(nums):
            res.append(permutation[:])  # 保存当前排列的副本
            return

        # 遍历所有数字
        for i in range(len(nums)):
            if not used[i]:  # 如果当前数字没有被使用
                used[i] = True  # 标记为已使用
                permutation.append(nums[i])  # 将数字加入当前排列
                self.backtrace(nums, used, permutation, res)  # 递归探索
                permutation.pop()  # 撤销选择
                used[i] = False  # 回溯时恢复未使用状态

    def permute(self, nums):
        res = []  # 用于存储所有结果
        used = [False] * len(nums)  # 标记每个数字是否被使用
```

```
        permutation = []  # 当前排列
        self.backtrace(nums, used, permutation, res)
        return res
```

Eg 4 N皇后

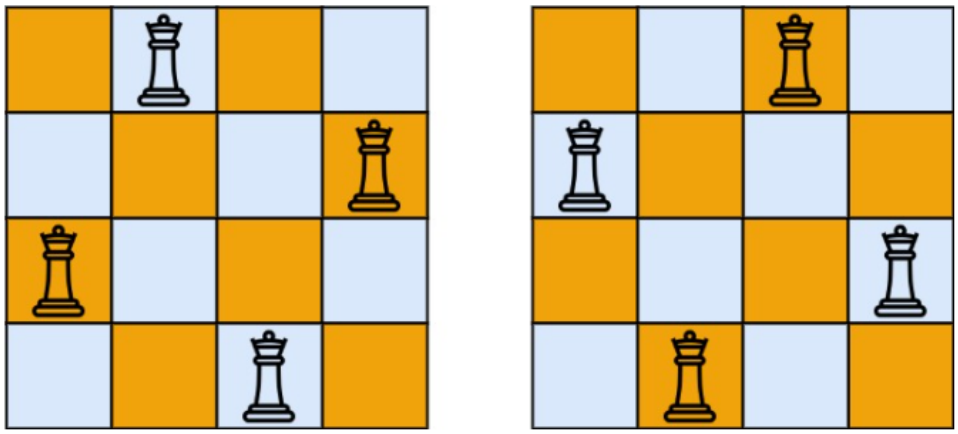
按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

n 皇后问题 研究的是如何将 `n` 个皇后放置在 `n×n` 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 `n` ，返回所有不同的 **n 皇后问题** 的解决方案。

每一种解法包含一个不同的 **n 皇后问题** 的棋子放置方案，该方案中 `'Q'` 和 `'.'` 分别代表了皇后和空位。

示例 1：



> **输入：** n = 4
> **输出：** [[".Q..","...Q","Q...","..Q."],
> ["..Q.","Q...","...Q",".Q.."]]
> **解释：** 如上图所示，4 皇后问题存在两个不同的解法。

解：使用回溯法解决。由于`n`个皇后必然处于不同的`n`行和`n`列，也即每一行必然有且仅有1个，因此逐行对棋盘进行构造（与回溯）。由于要求不同皇后位于不同行/列/主对角线/副对角线，其中一条主对角线上元素的共性为其`row-col`值相同，一条副对角线上元素的共行为其`row+col`值相同。

因此，类似上题全排列问题的做法，设置3个集合：`columns`、`diagonals1`、`diagonals2`，分别表示棋盘构造到当前行时，哪些列、主对角线、副对角线已经被占用（其中主对角线、副对角线集合中的元素为各种`row-col`的值和各种`row+col`的值，因为它们能够唯一标识某个主/副对角线）。主函数如下：

```
    def solveNQueens(self, n):
        res = []  # 存储所有解法
        board = [["."] * n for _ in range(n)]  # 初始化棋盘
        columns = set()  # 记录被占用的列
        diagonals1 = set()  # 记录被占用的主对角线 (row - col)
        diagonals2 = set()  # 记录被占用的副对角线 (row + col)
        self.backtrace(0, n, columns, diagonals1, diagonals2, board, res)
        return res
```

对于回溯递归函数，每次回溯针对一行进行，其逻辑如下：

- 若当前行数已经到达`n`，说明此时已经构造完成了一个棋盘，因此可以将当前棋盘加入`res`中，并返回。
- 否则，进一步开始在当前行中，对列进行循环，也即尝试将该行的皇后放到每个位置上：首先判断当前尝试的皇后位置所在列/主对角线/副对角线是否已经被用过了，如果用过了就直接跳过，如果没用过则可以尝试这个位置：

  尝试将该位置的皇后添加到棋盘中，并将当前位置所在的列/主对角线/副对角线都分别记入相应集合避免以后重复，然后对下一行进行递归回溯。最后，再将该位置的尝试撤回：将棋盘中的该位置复原，并在列/主对角线/副对角线集合中删除这个点，然后继续尝试该行的下一个位置...

```python
class Solution:
    def backtrace(self, row, n, columns, diagonals1, diagonals2, board, res):
        # 如果已经放置了 n 个皇后，保存当前棋盘
        if row == n:
            res.append(["".join(row) for row in board])  # 保存当前棋盘
            return

        # 尝试在当前行的每一列放置皇后
        for col in range(n):
            if col in columns or (row - col) in diagonals1 or (row + col) in diagonals2:
                continue  # 如果当前列或对角线被占用，跳过

            # 做选择
            board[row][col] = 'Q'
            columns.add(col)
            diagonals1.add(row - col)
            diagonals2.add(row + col)

            # 递归到下一行
            self.backtrace(row + 1, n, columns, diagonals1, diagonals2, board, res)

            # 撤销选择（回溯）
            board[row][col] = '.'
            columns.remove(col)
            diagonals1.remove(row - col)
            diagonals2.remove(row + col)

    def solveNQueens(self, n):
        res = []  # 存储所有解法
        board = [["."] * n for _ in range(n)]  # 初始化棋盘
        columns = set()  # 记录被占用的列
        diagonals1 = set()  # 记录被占用的主对角线 (row - col)
        diagonals2 = set()  # 记录被占用的副对角线 (row + col)
        self.backtrace(0, n, columns, diagonals1, diagonals2, board, res)
        return res
```

Eg 5. 组合总和

给你一个 **无重复元素** 的整数数组 `candidates` 和一个目标整数 `target` ，找出 `candidates` 中可以使数字和为目标数 `target` 的 所有 **不同组合** ，并以列表形式返回。你可以按 **任意顺序** 返回这些组合。

`candidates` 中的 **同一个** 数字可以 **无限制重复被选取** 。如果至少一个数字的被选数量不同，则两种组合是不同的。

解：

需要返回所有的组合，因此仍可以使用回溯法。在总函数中，首先将序列进行排序来便于处理，这样便于确认每次能加入组合的下一个元素有哪些（规定每个时刻的组合`combination`中的元素都是非递减的，避免重复）。

backtrace函数参数中，设置一个target变量，用于统计当前距离初始目标所差多少。在backtrace内部的循环有点类似于组合题，也是从当前`start`点开始尝试进一步加入后边的值。由于每个数字都可以被无限制选取，所以循环中`start`本身也作为下一个数字的可选位置进行试探（而不是`start+1`）。在循环中，若当前的数字已经大于了`target`，则可以跳过。递归的终止条件为target变为0。

在确认本次能够加入组合的下一个元素有哪些时，由于原始序列已经进行了排序，且规定了`combination`中的元素排列是非递减的，所以只有`>= start`的元素是有可能加入的，不能再加入比`start`还小的元素。另外，如果发现要加入的下一个元素本身就已经超过距离目标的剩余值了，那么也不能将它作为下一个元素加入组合。

```
class Solution(object):
    def backtrace(self, candidates, target, start, combination, res):
        # 终止条件：当目标值为 0 时，说明找到一个有效组合
        if target == 0:
            res.append(combination[:])  # 保存当前组合
            return

        # 遍历所有候选数字
        for i in range(start, len(candidates)):
            if target < candidates[i]:  # 如果当前数字大于目标值，跳过
                continue

            # 做选择：将当前数字加入组合
            combination.append(candidates[i])

            # 递归：继续从当前数字开始尝试（允许重复选择）
            self.backtrace(candidates, target - candidates[i], i, combination, res)

            # 撤销选择（回溯）
            combination.pop()

    def combinationSum(self, candidates, target):
        res = []  # 用于存储所有结果
        combination = []
        candidates.sort()  # 排序以便提前剪枝
        self.backtrace(candidates, target, 0, combination, res)
        return res
```

Eg6. 括号生成

数字 `n` 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 **有效的** 括号组合。

示例：

```
输入：n = 3
输出：["((()))","(()())","(())()","()(())","()()()"]
```

解：

生成类问题使用回溯法解决。本题每次探索时能进一步添加的候选元素只有`'('`或`')'`这两种，但由于最终目标是要构造出符合题意的有效串，因此构造子串的每一步时不能随意尝试`'('`和`')'`，需要确保尝试后构造出来的子串未来能够构成完整有效串。具体而言，一个子串需要具备如下两个条件，未来才有可能构成有效串：

- 任何时刻，左括号数量都不能超过最大数量限制`n`
- 任何时刻，右括号数量都不能超过左括号数量，否则会出现`')('`, `'())'`这种无效情况，一旦左括号少了未来是救不回来的

因此，需要持续记录当前左括号和右括号分别已经使用了多少个，从而依据上述条件来确认本步是否能探索添加`'('`和`')'`，如果能的话就进一步调用回溯函数来探索一下

```
class Solution:
    def backtrace(self, cur_str, n, left_used, right_used, results):
        if left_used == n and right_used == n:  # 此时已经用了n对括号，说明已经组合成了一个完整有效串，将其记录在结果中
            results.append(cur_str)
            return

        # 一个子串未来能够生成有效串需要满足的条件：
        # 1. 任何时候左括号数量不能超过最大限度n
        # 2. 任何时候右括号数量不能超过左括号数量，否则会出现')(', '())'这种无效情况，一旦左括号少了未来是救不回来的
        # 因此，根据这两个条件，继续探索下一步可以添加的括号
        # 下边这两种情况分别探索了加'('或者加')'的情况，只有可能加这两种括号

        if left_used < n:    # 如果当前左括号数量<n，那说明还可以继续加左括号，因此尝试一下继续加一个左括号
            cur_str = cur_str + '('
            self.backtrace(cur_str, n, left_used+1, right_used, results)
            # 然后撤销添加'('的这一步
            cur_str = cur_str[:-1]

        if right_used < left_used:  # 如果当前右括号数量小于左括号数量，那说明还可以继续加右括号，因此尝试一下继续加一个右括号
            cur_str = cur_str + ')'
            self.backtrace(cur_str, n, left_used, right_used+1, results)
            # 然后撤销添加')'的这一步
            cur_str = cur_str[:-1]


    def generateParenthesis(self, n: int) -> List[str]:
```

```
        results = []
        cur_str = ""
        left_used = 0
        right_used = 0
        self.backtrace(cur_str, n, left_used, right_used, results)
        return results
```

Eg 7. 电话号码的字母组合

给定一个仅包含数字 `2-9` 的字符串，返回所有它能表示的字母组合。答案可以按 **任意顺序** 返回。

给出数字到字母的映射如下（与电话按键相同），注意 1 不对应任何字母：

```
digit_map = {
    '2': ['a', 'b', 'c'],
    '3': ['d', 'e', 'f'],
    '4': ['g', 'h', 'i'],
    '5': ['j', 'k', 'l'],
    '6': ['m', 'n', 'o'],
    '7': ['p', 'q', 'r', 's'],
    '8': ['t', 'u', 'v'],
    '9': ['w', 'x', 'y', 'z']
}
```

示例：

```
输入：digits = "23"
输出：["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

解：

本题比较简单，每遍历到一个数时只需依次尝试它对应的几个字母即可

```
class Solution:
    def backtrace(self, digits, digit_map, i, combination, res):
        if i == len(digits):
            res.append(''.join(combination))    # ['a', 'd'] -> 'ad'
            return

        # 依次尝试第i个数字对应的几个字母
        for letter in digit_map[digits[i]]:
            combination.append(letter)
            self.backtrace(digits, digit_map, i+1, combination, res)
            combination.pop()

    def letterCombinations(self, digits: str) -> List[str]:
        if not digits: return []

        digit_map = {
            '2': ['a', 'b', 'c'],
```

```
            '3': ['d', 'e', 'f'],
            '4': ['g', 'h', 'i'],
            '5': ['j', 'k', 'l'],
            '6': ['m', 'n', 'o'],
            '7': ['p', 'q', 'r', 's'],
            '8': ['t', 'u', 'v'],
            '9': ['w', 'x', 'y', 'z']
        }

        res = []
        combination = []
        self.backtrace(digits, digit_map, 0, combination, res)

        return res
```

贪心算法

> 特点在于：通过某种规则，每次只处理一个元素，而不需要顾及全局状态，最终将所有元素处理完时，恰好就能达到全局最优。

Eg1. 根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 `people` 表示队列中一些人的属性（不一定按顺序）。每个 `people[i] = [hi, ki]` 表示第 `i` 个人的身高为 `hi` ，前面 正好 有 `ki` 个身高大于或等于 `hi` 的人。

请你重新构造并返回输入数组 `people` 所表示的队列。返回的队列应该格式化为数组 `queue` ，其中 `queue[j] = [hj, kj]` 是队列中第 `j` 个人的属性（`queue[0]` 是排在队列前面的人）。

示例：

```
输入：people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
输出：[[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
解释：
编号为 0 的人身高为 5 ，没有身高更高或者相同的人排在他前面。
编号为 1 的人身高为 7 ，没有身高更高或者相同的人排在他前面。
编号为 2 的人身高为 5 ，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。
编号为 3 的人身高为 6 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
编号为 4 的人身高为 4 ，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。
编号为 5 的人身高为 7 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
因此 [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]] 是重新构造后的队列。
```

解：本题本质上是使用贪心算法：因为矮的人不影响高的人在队列中的“位置”（每个人的位置信息是前边有多少个人比他高，因此在最终的队列中不管前边插入多少个比他矮的人都不会影响他所站的位置是否符合其位置信息），所以可以从高到矮将人插入队列（也即，先把高的人按照其“位置”信息安排妥当，后边随便再插入多少矮的人都不会造成破坏了）。因此，首先根据身高进行排序，然后逐个按照每个人的位置信息将其插入列表中

```python
class Solution:
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        # 按身高降序排序，同身高则按k升序排序
        people.sort(key=lambda x: (-x[0], x[1]))
        queue = []
        for p in people:
            queue.insert(p[1], p)   # 将p插入到当前队列的第p[1]个位置
        return queue
```

Eg2. 分发糖果

`n` 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。

你需要按照以下要求，给这些孩子分发糖果：

- 每个孩子至少分配到 `1` 个糖果。
- 相邻两个孩子评分更高的孩子会获得更多的糖果。

请你给每个孩子分发糖果，计算并返回需要准备的 **最少糖果数目** 。

解：

本题也可以使用贪心算法解决：先进行一次从左到右的遍历，来使得每个孩子左边的大小关系得到满足；然后再进行一次从右到左的遍历，使得每个孩子右边的大小关系得到满足。

具体而言，初始化每个孩子的糖果数量为1。在从左到右遍历中，如果发现当前孩子的得分大于他左侧的孩子，那么就将他的糖果数量设为左边孩子的糖果数量+1，这样在遍历结束后就可以确保每个孩子左侧的大小关系得到满足；然后进行一次从右到左的遍历，来在当前基础上对于某些不满足右侧大小关系的情况进行修正，如果发现当前孩子的得分大于他右边孩子，但糖果数量不大于右边孩子，则将当前孩子的糖果数量设为右边孩子糖果数量+1。

如此进行完两次遍历后，即可确保每个孩子的左右大小关系均得到满足，且因为尽可能缩小了相邻孩子的糖果数量差异，使得总糖果数量最小。

```python
class Solution:
    def candy(self, ratings: List[int]) -> int:
        # 该列表保存每个孩子的糖果数量
        candies = [1 for _ in range(len(ratings))]

        # 首先，从左到右遍历，顾及每个人左侧的大小关系
        # 如果当前孩子的分数比它左边孩子的分数高，那么给这个孩子的糖果数量就是它左边孩子的数量+1；否则就只给他1个糖果
        for i in range(1, len(ratings)):
            if ratings[i] > ratings[i-1]:
                candies[i] = candies[i-1] + 1
            else:
                candies[i] = 1

        # 然后，从右到左遍历，顾及每个人右侧的大小关系
        # 如果当前孩子的分数比它右边孩子的分数高，且他目前分配的糖果数量不大于右边的孩子，那么就将它的糖果数量设置为右边孩子糖果数量+1；否则保持原状
        for i in range(len(ratings)-2, -1, -1):
```

```
            if ratings[i] > ratings[i+1] and candies[i] <= candies[i+1]:
                candies[i] = candies[i+1] + 1

        return sum(candies)
```

Eg3. 任务调度器

给你一个用字符数组 `tasks` 表示的 CPU 需要执行的任务列表，用字母 A 到 Z 表示，以及一个冷却时间 `n`。每个周期或时间间隔允许完成一项任务。任务可以按任何顺序完成，但有一个限制：两个 **相同种类** 的任务之间必须有长度为 `n` 的冷却时间。

返回完成所有任务所需要的 **最短时间间隔** 。

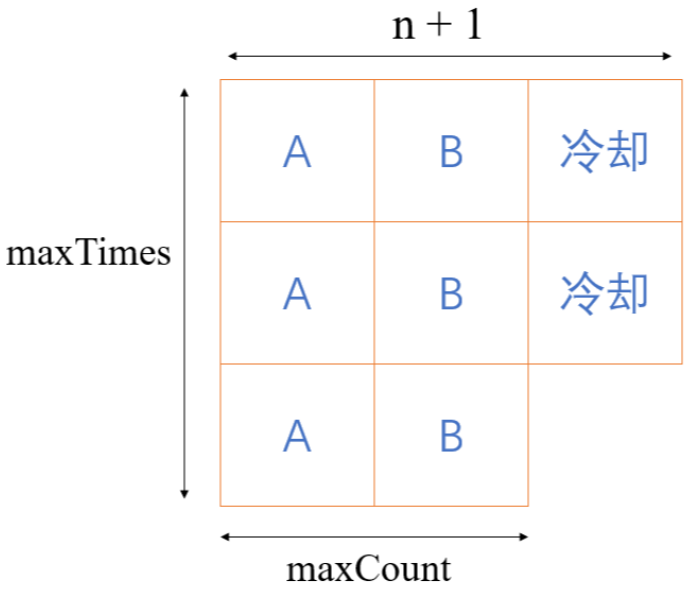
示例：

```
输入：tasks = ["A","A","A","B","B","B"], n = 2

输出：8

解释：

在完成任务 A 之后，你必须等待两个间隔。对任务 B 来说也是一样。在第 3 个间隔，A 和 B 都不能完成，所以你需要待命。在第 4 个间隔，由于已经经过了 2 个间隔，你可以再次执行 A 任务。


输入：tasks = ["A","C","A","B","D","B"], n = 1

输出：6

解释：一种可能的序列是：A -> B -> C -> D -> A -> B。

由于冷却间隔为 1，你可以在完成另一个任务后重复执行这个任务。
```

解：

一种贪心方法为：先安排好出现频率最高的任务，使得它们各自的间隔为`n`（例如下图中的`'A'`任务），如果还有其他任务出现频率也一样高的话（例如下图`'B'`），则首先将其紧挨着`'A'`插入空档中。然后，剩下的空档则可以插入其他的任务。

当任务种类数量比较少的时候，也即最终排列好后也还存在冷却空档时（如下图情况），可见总时间即为下图面积：其中`maxTimes`为出现频率最高的任务出现的次数，`maxCount`为共有多少个任务为出现频率最高的任务，则面积为：`(maxTimes-1) * (n+1) + maxCount`。



当任务种类数量比较多的时候，可以把所有空档都塞满（甚至冷却空档无法完全容纳所有任务，还得在最后额外加点时间来跑完剩余的任务），此时总时间就是任务数量`len(tasks)`，不存在中间空档时间浪费。如果还按照上述公式计算的话，则会漏掉最后额外加的时间（也即只统计了冷却空档中塞的任务，漏掉了溢出的任务），此时计算结果会比实际结果`len(tasks)`小一些。

综上，为了综合考虑存在空档/溢出的情况，在最终结果中给二者取max即可：

```
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        # 首先统计每种任务出现的次数，并找到出现次数最大的任务（们），并将它们记录
        task_appear_num = {}
        max_appear_tasks = []
        max_appear_num = 0
        for i in range(65, 91):
            task_appear_num[chr(i)] = 0
        for task in tasks:
            task_appear_num[task] += 1
            if task_appear_num[task] > max_appear_num:
                max_appear_tasks = [task]
                max_appear_num = task_appear_num[task]
            elif task_appear_num[task] == max_appear_num:
                max_appear_tasks.append(task)

        # 此时可以得到：
        # 最多的任务出现的次数
        maxTimes = max_appear_num
        # 共有一个最多的任务
        maxCount = len(max_appear_tasks)

        # 当存在冷却空档的情况下，按照如下公示即可算出总时间
        all_time = (maxTimes - 1) * (n+1) + maxCount

        # 如果任务种类比较多导致不需要冷却空档，都可以填满，则总时间应为len(tasks)，
        # 此时如果还按照上式算的话会导致冷却空档溢出（不足以承载那些tasks数量），算出来的会偏小一点
        # 因此，最终答案只需把公式结果和len(tasks)取二者中的较大值即可
```

```
        all_time = max(all_time, len(tasks))

        return all_time
```

Eg4. 跳跃游戏

给你一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标，如果可以，返回 `true` ；否则，返回 `false` 。

示例：

```
输入：nums = [2,3,1,1,4]
输出：true
解释：可以先跳 1 步，从下标 0 到达下标 1，然后再从下标 1 跳 3 步到达最后一个下标。

输入：nums = [3,2,1,0,4]
输出：false
解释：无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。
```

解：

由于跳跃区间的覆盖是连续的（因为数组元素值表示的是在该位置可以跳跃的“最大”长度），也即如果能到达某个点的话，那么它前边的所有点都可到达。因此从前往后遍历数组的过程中，可以只维护一个当前最远可达位置，每遍历到一个位置`i`时，如果它小于等于当前最远可达位置（说明它本身可达），则从它出发最远能到达`i + nums[i]`，因此可以用这个值更新最远可达位置。若某个时刻发现当前最远可达位置超过了数组长度，那么说明末尾元素可达，直接返回`True`；若某时刻发现当前位置不可达，那么它后边所有元素都不可达了，此时直接返回`False`

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        if len(nums) == 1:
            return True

        # 维护当前能到达的最远点。由于跳跃覆盖区间是连续的，因此最远点及之前所有点都能到达
        max_reachable = 0 + nums[0]
        for i in range(1, len(nums)):
            if i <= max_reachable:  # 如果发现当前点小于等于当前最远可达点，说明当前点可达，可以进一步用该点的信息更新最远可达点
                max_reachable = max(max_reachable, i + nums[i])
                if max_reachable >= len(nums) - 1:  # 如果某时刻发现最远可达点超过了末尾点，则说明末尾点可达，可以返回True
                    return True
            else:   # 如果发现从某个点开始不可达了，那么它之后的点都没法到达，因此最后一个点也不可达
                return False
```

Eg5. 加油站

在一条环路上有 `n` 个加油站，其中第 `i` 个加油站有汽油 `gas[i]` 升。

你有一辆油箱容量无限的的汽车，从第 `i` 个加油站开往第 `i+1` 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组 `gas` 和 `cost` ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 `-1` 。如果存在解，则 **保证** 它是 **唯一** 的。

解：

最简单的思路即为依次尝试从每个点作为起始点开始往右走（通过对`n`取模来处理回环情况）。在以每个点为起点往右走的过程中，维护一个当前剩余汽油量，每走到一个点时一方面补充该点的加油量，另一方面损耗掉从该点到其下一个点之间需要的汽油量，若发现此时剩余的汽油量为负数则说明到不了下一个点了。如果发现能够成功往右走`n`步，则说明从这个起点开始可以走一圈。如果发现所有起点都不能往右走`n`步则说明不存在解。

进一步可以考虑剪枝：设从`i`点开始往右走最远能到`j`，但无法走完一圈，则可以断言：以`[i,j]`之间的所有点出发都不可能走完一圈。因为对于这之间的任何一个点`k`来说，从`i`出发经过它并补充这里的汽油之前，汽油的余量必然是>=0的，要不然没法从`i`到`k`，可见到达`k`时一定是"带资进组"的，在这种情况下继续往后走到`j`都会面临汽油不足的问题，那么从`k`开始走相当于"白手起家"，条件不会优于从`i`出发时的情况，因此最远也不可能走过`j`。因此，尝试完`i`作为起点并发现最远能到达`j`后，下一次尝试起点时可以直接略过它们之间的所有点，直接从`j+1`开始继续尝试（实际实现时需要考虑回环问题，因此不能直接设置`i=j+1`，详见代码）

```
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        i = 0       # i为本次尝试的起点
        n = len(gas)

        while i < n:    # i为本次尝试的起点，从0开始尝试，直到到n-1
            gas_left = 0    # 当前剩余汽油量

            cnt = 0     # 当前（从左往右）距离起点i的距离,当达到n时说明绕回了一圈
            while cnt < n:
                j = (i + cnt) % n   # j为当前位置
                gas_left += gas[j]  # 在当前位置补充汽油
                gas_left -= cost[j] # 从当前位置到下一个位置损耗汽油
                if gas_left < 0:    # 到下一个位置时剩余汽油为负数，说明实际上到不了下一个位置了，j就是从i出发最远能到达的位置
                    break
                else:       # 能到下一个位置，继续往下走一步
                    cnt += 1
                    if cnt == n:    # 如果此时发现已经能从i往右走n格的距离，那就说明绕了一圈，i就是可行的起始点
                        return i

            # 以i开始最远能到j，因此[i,(n),j]之间的点出发都不可能绕一圈，下一个起点从j+1开始尝试
            # 由于j+1有可能是越过末尾点后循环回来的位置，因此直接设i=j+1有可能造成重复尝试之前已经试过的起点。因此关注从i到j+1从左往右的距离cnt+1，将i右移cnt+1
            # 如果移动后i超过了末尾点则说明所有可能的起始点都试完了，此时while i < n循环退出并返回-1
            i += cnt + 1
```

```
        # 最终也没找到能绕一圈的起始位置，返回-1
        return -1
```

位运算

1、只出现一次的数字

给你一个 **非空** 整数数组 `nums` ，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。

你必须设计并实现线性时间复杂度的算法来解决此问题，且该算法只使用常量额外空间。

解：由于要求使用常数额外空间，因此不能使用哈希表来记录每个元素出现的次数。

可以使用位运算中的异或（XOR，python中用`^`符号计算）来解决。异或有性质：1）任何数和自己的异或都为0：`x^x=0`；2）任何数和0的异或都是自身：`x^0=x`；3）异或满足交换律和结合律

由于异或满足交换律，因此可以假设列表中各个出现两次的元素都挨在一起，单独的元素处于最后：`[a,a,b,b,c,c,...,x]`，然后从头到尾将列表中所有元素都做一遍异或：`a^a^b^b^...^x`，根据结合率可知所有成对的元素都将被消成0，最后只剩下`0^x=x`，即可得到单独的元素。

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        x = 0
        for num in nums:
            x ^= num
        return x
```

2、汉明距离

两个整数之间的汉明距离指的是这两个数字对应二进制位不同的位置的数目。

给你两个整数 `x` 和 `y`，计算并返回它们之间的汉明距离。

解：

首先使用异或操作得到`s=x^y`，则`s`的二进制表示中为1的位置都是x,y中不同的一位

因此，只需求一下`s`中有多少个1即可，具体方法为：不断使用右移一位操作`s >>= 1`，每移一次就统计一下当前的末位是不是1，方法为和1取and：`s & 1`

```python
class Solution:
    def hammingDistance(self, x: int, y: int) -> int:
        s = x ^ y   # x,y的二进制表示中不同的位在s中都是1
        num_1 = 0
        while s:
            num_1 += s & 1  # 获得s的最低位
            s >>= 1 # 将s右移一位
        return num_1
```

3、比特位计数

给你一个整数 `n` ，对于 `0 <= i <= n` 中的每个 `i` ，计算其二进制表示中 `1` 的个数 ，返回一个长度为 `n + 1` 的数组 `ans` 作为答案。

示例：

```
输入：n = 2
输出：[0,1,1]
解释：
0 --> 0
1 --> 1
2 --> 10
```

解：对于每个数字，只需不断将其和1做`&`操作来判断其此时的最低位是否为1，然后再右移一位，即可统计出其二进制表示中所有的1的数量

```python
class Solution:
    def countBits(self, n: int) -> List[int]:
        result = []
        for i in range(n+1):
            tmp_i = 0   # i的二进制表示中1的个数
            while i:
                tmp_i += i & 1
                i >>= 1
            result.append(tmp_i)
        return result
```

其他

1、除自身以外数组的乘积

给你一个整数数组 `nums`，返回 数组 `answer` ，其中 `answer[i]` 等于 `nums` 中除 `nums[i]` 之外其余各元素的乘积 。

题目数据 **保证** 数组 `nums` 之中任意元素的全部前缀元素和后缀的乘积都在 **32 位** 整数范围内。

请 **不要使用除法，**且在 `O(n)` 时间复杂度内完成此题。

```
示例:
输入: nums = [1,2,3,4]
输出: [24,12,8,6]
```

解:

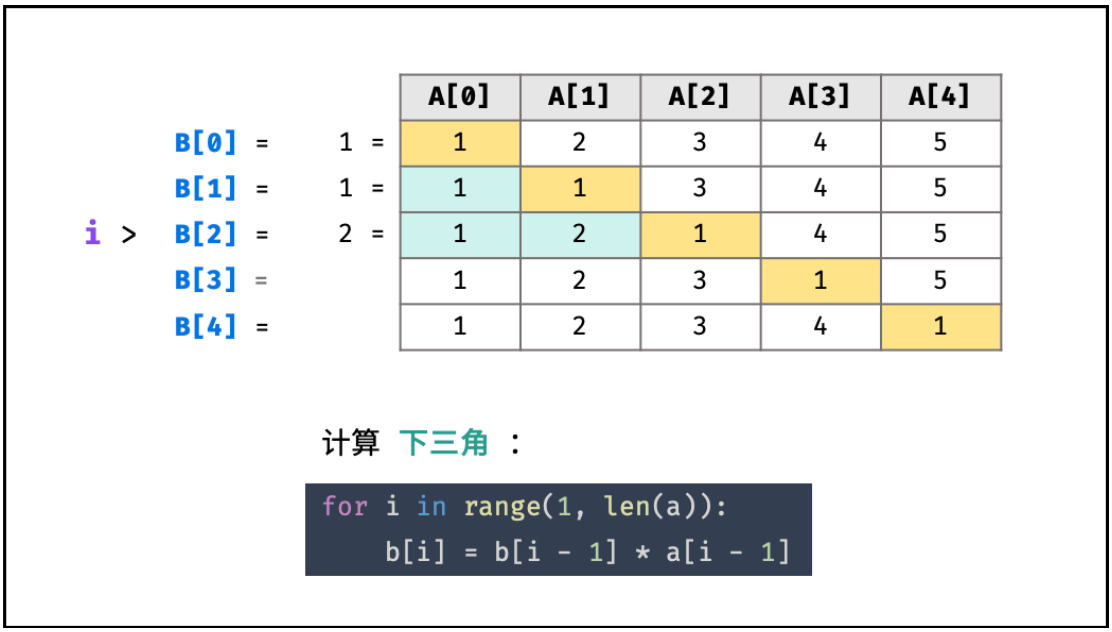
如果在外层for循环遍历每个元素，在内层for循环遍历该元素以外的所有元素并累乘，则会造成$O(n^2)$的复杂度。

事实上，可以发现各个元素以外的其他元素的累乘存在很大部分的重合，如：`answers[2]=nums[0]*nums[1]*nums[3]*..., answers[3]=nums[0]*nums[1]*nums[2]*nums[4]*...`，只关注左半部分的话即可发现`answers[2],answers[3]`共享了`nums[0]*nums[1]`这部分，差距仅在于`nums[2]`。

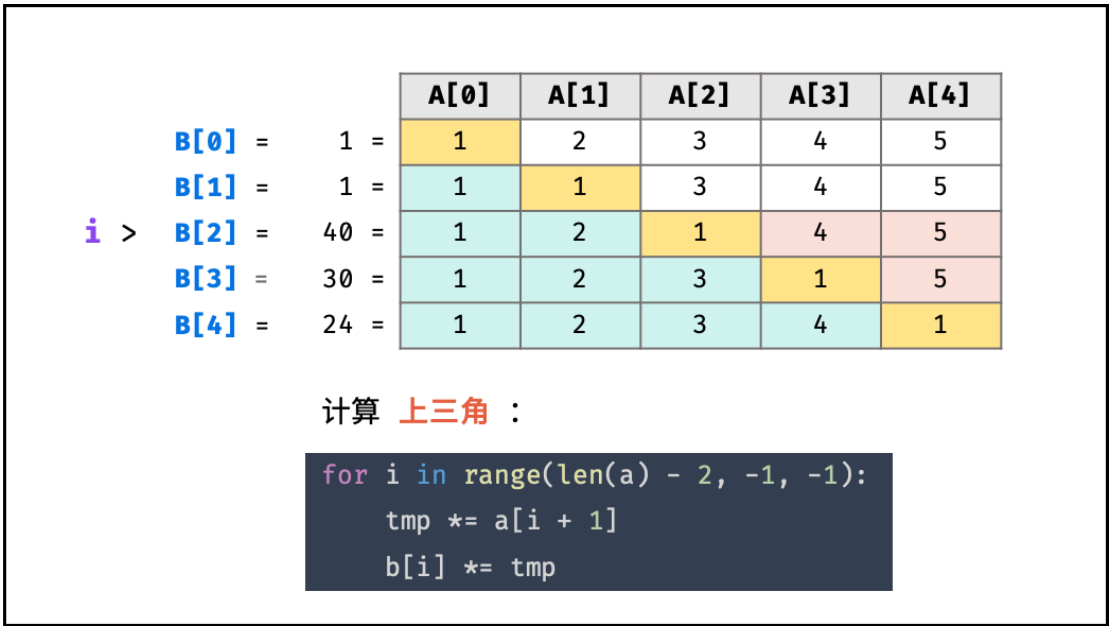
因此，先考虑左半部分的情况下，可以通过递推的方式利用前边的累乘结果，从而求取`answers[i]`

```
answers[i] = answers[i-1] * nums[i-1]
```

这样使用一轮从头到尾的for循环来遍历各个元素，即可得到各个元素左半部分的累乘结果。



类似地，得到左半部分的累乘结果后，可以进一步从右到左遍历一遍，逐次累积得到各个元素右半部分的累乘结果。具体而言，得到某个元素的右半部分累乘结果后，将其乘到第一轮循环中得到的左半部分累乘结果`answers[i]`中即可得到该元素的总累乘结果，而右半部分累乘结果在后续的唯一用途就是用于推导出下一个元素的右半部分累乘结果，其本身并不需要被记录，因此可以用一个数字`tmp_answers`来记录当前的右半部分累乘结果



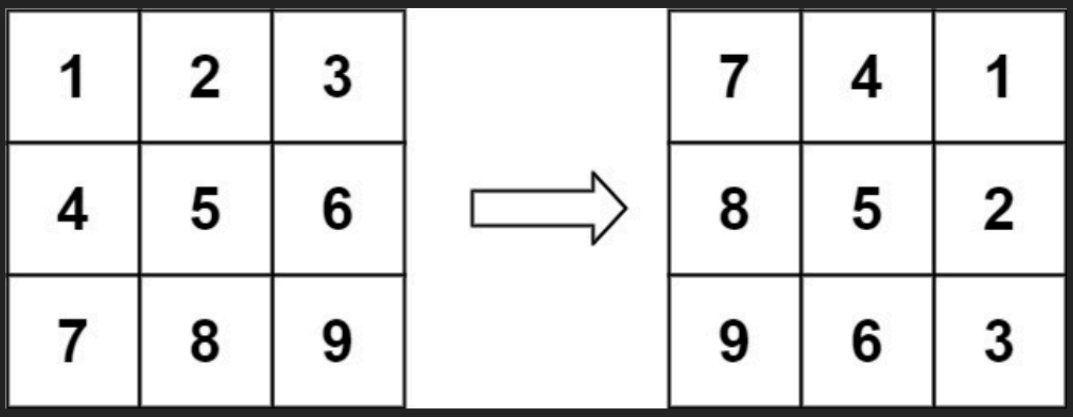

```python
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        answers = [1] * len(nums)
        for i in range(1, len(nums)):
            answers[i] = answers[i-1] * nums[i-1]

        tmp_answers = 1
        for j in range(len(nums)-2, -1, -1):
            tmp_answers = tmp_answers * nums[j+1]
            answers[j] = answers[j] * tmp_answers
        return answers
```

2、旋转图像

给定一个 *n* × *n* 的二维矩阵 `matrix` 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。**请不要** 使用另一个矩阵来旋转图像。

示例 1：

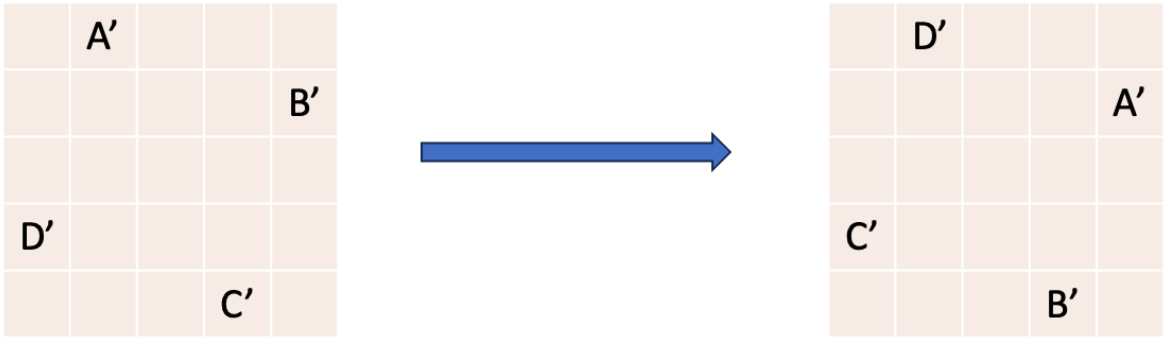


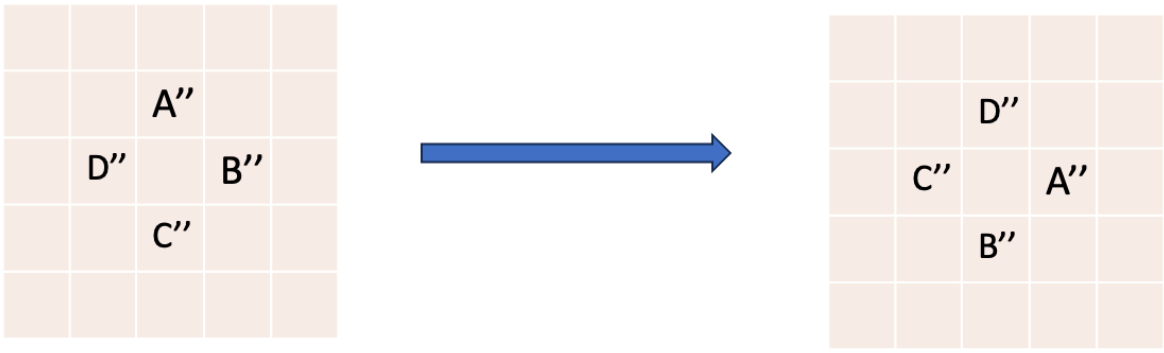
输入： `matrix = [[1,2,3],[4,5,6],[7,8,9]]`
输出： `[[7,4,1],[8,5,2],[9,6,3]]`

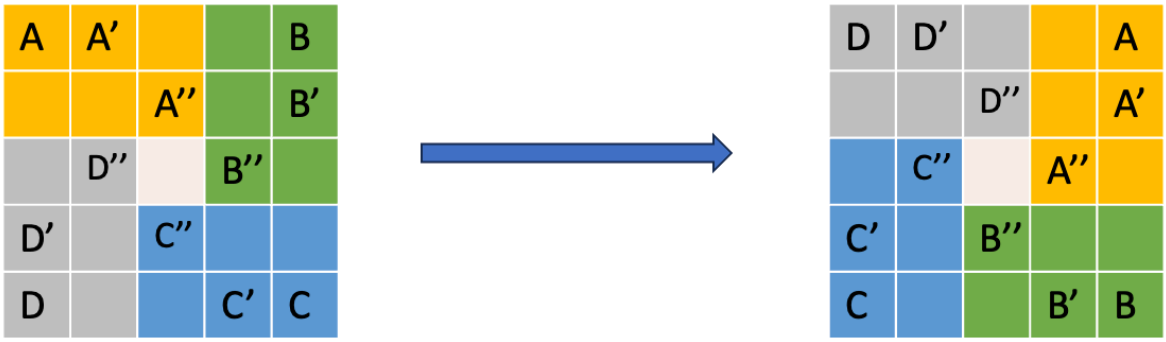
解：

观察发现，每个元素的旋转都会总共涉及其所在的“一圈”的4个元素，如下图所示，替换顺序为：`D -> A`，`A -> B`，`B -> C`，`C -> D`，因此，只要把左上角1/4区域（$0 \leq i \leq n//2$，$0 \leq j \leq (n+1)//2$）的所有元素所在的圈都处理好，就可以覆盖掉整个矩阵中所有需要旋转的元素。

接下来考虑某一组ABCD，设`A=matrix[i][j]`，考虑BCD的对应坐标：B所在列数从右数就是A所在行数，B所在行数就是A所在列数，因此`B=matrix[j][n-i-1]`；C和A完全中心对称，因此`C=[n-i-1][n-j-1]`；D所在行数从下数就是A所在列数，D所在列数就是A所在行数，因此`D=matrix[n-j-1][i]`。然后即可遍历左上角1/4块的各个A元素，然后依据上述替换顺序（`D -> A`，`A -> B`，`B -> C`，`C -> D`）依次替换即可。注意由于这4个元素之间存在覆盖关系，因此需要使用tmp来在轮换过程中临时保存某个元素值，下文代码中为了清晰起见每次都使用`A,B,C,D`为当前圈上的元素保存了一份副本，因此不再需要额外使用tmp了。这样的话需要的额外空间大小总是O(1)的。








```python
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)
        # 对于 0 <= i < n // 2, 0  <= j < (n+1) // 2 也即左上1/4的子矩阵中的各个元素分别做一轮旋转即可
        # 每一轮中的相对位置：A: matrix[i][j], B: matrix[j][n-i-1], C: matrix[n-i-1][n-j-1], D: matrix[n-j-1][i]
        for i in range(n//2):
            for j in range((n+1)//2):
                A = matrix[i][j]
                B = matrix[j][n-i-1]
                C = matrix[n-i-1][n-j-1]
                D = matrix[n-j-1][i]
                # D -> A
                matrix[i][j] = D
                # C -> D
```

```
                matrix[n-j-1][i] = C
                # B -> C
                matrix[n-i-1][n-j-1] = B
                # A -> B
                matrix[j][n-i-1] = A
```

3、下一个排列

整数数组的一个 **排列** 就是将其所有成员以序列或线性顺序排列。

- 例如，`arr = [1,2,3]`，以下这些都可以视作 `arr` 的排列：`[1,2,3]`、`[1,3,2]`、`[3,1,2]`、`[2,3,1]`。

整数数组的 **下一个排列** 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 **下一个排列** 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

- 例如，`arr = [1,2,3]` 的下一个排列是 `[1,3,2]`。
- 类似地，`arr = [2,3,1]` 的下一个排列是 `[3,1,2]`。
- 而 `arr = [3,2,1]` 的下一个排列是 `[1,2,3]`，因为 `[3,2,1]` 不存在一个字典序更大的排列。

给你一个整数数组 `nums`，找出 `nums` 的下一个排列。

必须原地修改，只允许使用额外常数空间。

解：

考虑一个例子：`[1,3,5,4,2]`

首先，从右往左遍历数组，然后查看包括当前数字在内的其右边的子串：`[2], [4,2], [5,4,2]` 这几个子串由于都是单调递减的（也即子串中每个数字都不小于其右侧数字），因此它们都没办法再“增大”了，它们的下一个排列将是翻转为单调递增后的第1个排列。

直到遍历到`[3,5,4,2]`时，此时第一次出现左侧比右侧数字小的情况（`3<5`），因此这个子串不是单调递减的，它是可以继续“增大”，因此考虑它的下一个排列。由于`3`右侧的子串`[5,4,2]`已经最大了，因此下一个排列要在`3`这个位置上“进位”为右侧子串中第一个大于`3`的数字，然后再将右侧子串翻转为单调递增的最小序列，即找到了下一个排列。

具体而言，重新从右到左遍历这个子串，并找到第一个大于当前数字的数字，在该例中为4（`4>3`），然后将`3,4`这两个数字进行调换，此时该子串变为`[4,5,3,2]`，也即完成了`3->4`的进位。进位后新的右侧子串`[5,3,2]`仍然是单调递减的，因此将它翻转变成`[2,3,5]`后就是它这一部分的下一个排列，此时即可得到原先`[3,5,4,2]`（也即以`3`为开头的最后一个序列）的下一个排列，也即以`4`作为开头的第1个序列`[4,2,3,5]`。

本质上，也即从右到左找到第1个可以“进位”的子串，然后将其“进位”，本例也即把`3`开头的最后一个子序列`[3,5,4,2]`进位为了以`4`开头的第一个子序列`[4,2,3,5]`，同时保持更高位的数字不变，也就得到了整个序列的下一个排列

```
class Solution:
    def reverse_list(self, nums, left, right):
        """
        原地翻转nums[left: right]（闭区间）这一段的元素
```

```
        """
        while left < right:
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

    def nextPermutation(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        n = len(nums)
        if n == 1: return nums

        # 首先，从右向左遍历，找到第一个小于其右侧相邻数字的数字
        found = False
        for i in range(n-2, -1, -1):
            if nums[i] < nums[i+1]:     # nums[i]即为第一个这样的数字
                found = True
                break
        if not found:   # 如果没有找到这样的数字，说明原始序列就是单调递减的，它本身已经是最后一个排列，因此下一个排列是第1个排列（单调递增），也即nums整体的翻转
            self.reverse_list(nums, 0, n-1)
            return

        # 然后，再重新从右向左遍历一次，找到nums[i]右侧第一个比它大的元素（nums[i]右侧的部分一定是单调递减的）
        for j in range(n-1, i, -1):
            if nums[j] > nums[i]:   # nums[j]就是nums[i]右侧第一个比它大的元素
                break
        # 然后将nums[i]和nums[j]交换位置
        nums[i], nums[j] = nums[j], nums[i]

        # 此时新的nums[i]右侧的部分仍为单调递减的，因此这部分本身是“最后一个排列”，将这部分翻转即得到了这部分的“下一个排列”（实际上也即这部分的第1个排列，变成单调递增）
        self.reverse_list(nums, i+1, n-1)
```

4、合并区间

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]` 。请你合并所有重叠的区间，并返回 *一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间* 。

示例：

```
输入：intervals = [[1,3],[2,6],[8,10],[15,18]]
输出：[[1,6],[8,10],[15,18]]
解释：区间 [1,3] 和 [2,6] 重叠，将它们合并为 [1,6].
```

解：

首先按照所有区间的起始位置将它们进行排序，然后以第一个区间作为当前构造区间的初始化，逐个遍历之后的区间：若发现下一个区间的起始位置大于当前构造区间的末尾位置，则说明二者没有重叠，将当前构造区间记录后，再将下一个区间作为当前构造区间并继续往后遍历；如果下一个区间的起始位置小于等于当前构造区间的末尾位置，则二者有重叠，再根据二者的末尾位置的大小关系来更新二者重叠后的新的当前区间。

关键在于将区间起始位置排序后再遍历，可以确保收集到的各个重叠区间也都是从小到大的，避免遗漏和重复等。

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        results = []    # 收集各个区间的重叠结果

        # 首先，按照每个区间的首项进行排序
        intervals.sort(key=lambda x:x[0])

        # 将当前构造的区间初始化为第一个区间
        cur_list = intervals[0]
        for sub_list in intervals[1:]:
            if sub_list[0] > cur_list[-1]:      # 下一个区间与当前区间没有重叠，则将当前区间记录到结果中，并将当前区间初始化为下一个区间
                results.append(cur_list)
                cur_list = sub_list
            else:   # 若下一个区间左端小于等于当前区间右端，则说明当前区间和下一个区间有重叠
                if sub_list[-1] <= cur_list[-1]:    # 如果下一个区间的右端小于等于当前区间的右端，由于下一个区间左端肯定大于等于当前区间的左端，因此下一个区间被当前区间包含了，重叠后还是当前区间
                    continue
                else:       # 如果下一个区间的右端大于当前区间右端，则二者合并后的右端应该更新为下一个区间的右端，从而扩大当前区间
                    cur_list[-1] = sub_list[-1]

        results.append(cur_list)    # 循环结束后，将最后一个当前区间也加入到结果中
        return results
```

5、H指数

给你一个整数数组 `citations` ，其中 `citations[i]` 表示研究者的第 `i` 篇论文被引用的次数。计算并返回该研究者的 **`h` 指数**。

根据维基百科上 h 指数的定义：`h` 代表“高引用次数”，一名科研人员的 `h` **指数** 是指他（她）至少发表了 `h` 篇论文，并且 **至少** 有 `h` 篇论文被引用次数大于等于 `h` 。如果 `h` 有多种可能的值，**`h` 指数** 是其中最大的那个。

示例：

```
citations = [3,0,6,1,5]
输出：3
解释：给定数组表示研究者总共有 5 篇论文，每篇论文相应的被引用了 3, 0, 6, 1, 5 次。
     由于研究者有 3 篇论文每篇 至少 被引用了 3 次，其余两篇论文每篇被引用 不多于 3 次，所以她的 h 指数是 3。
```

解：

首先将`citations`列表从大到小倒序排列，并初始化`h=0`，然后从大到小开始遍历各个`citation`：如果这个`citation>=h+1`，则说明找到了一篇引用量至少是`h+1`的文章，且这个`citation`前边的那些文章引用量一定也至少是`h+1`，因此可以把`h->h+1`，如此迭代直到发现`citation`已经无法大于等于当前的`h+1`为止。

```
class Solution:
    def hIndex(self, citations: List[int]) -> int:
        citations = sorted(citations, reverse=True)
        h = 0

        for citation in citations:
            if citation >= h+1:
                h += 1
        return h
```

数学问题

最大公约数和最小公倍数

- 最大公约数（gcd）：

  欧几里得辗转相除法求两个数的最大公约数：

  ```
  def gcd_two(a, b):
    while b != 0:
      a, b = b, a % b
     return abs(a)
  ```

  也可以直接用`math.gcd()`来计算两个数的最大公约数：`math.gcd(a, b)`

- 最小公倍数（lcm）：

  利用最大公约数间接求出，等于二者乘积除以二者最大公约数：`lcm(a, b) = |a * b| / gcd(a, b)`

  ```
  def lcm_two(a, b):
    return abs(a * b) // gcd_two(a, b)
  ```

求若干个数的最大公约数和最小公倍数：

只需依次对相邻的每两个数字求gcd、lcm即可（合并后的两个数字作为一个新数字，继续和下一个数字做运算）。可以使用`functools.reduce()`高效实现：

```python
from functools import reduce
from math import gcd

# 最大公约数
def gcd_list(nums):
  return reduce(gcd, nums)

# 最小公倍数
def lcm_list(nums):
  return reduce(lcm_two, numbers)
```

进制问题

- 10进制正整数->k进制数

  设十进制数$n = d_m k^m + \cdots + d_1 k^1 + d_0 k^0$，则其$k$进制表示为：$d_m d_{m-1} \cdots d_1 d_0$，其中$d_i \in \{0, 1, \cdots, k-1\}$。

  可以通过不断除以k并收集余数来从低位到高位依次得到各个系数$d_0, \cdots, d_m$，最后翻转一下就是k进制表示：

$$
\begin{gathered}
n//k = d_m k^{m-1} + \cdots + d_1,\ \ n\%k = d_0 \\
n = n//k \\
n//k = d_m k^{m-2} + \cdots + d_2, n\%k = d_1 \\
n = n//k \\
\vdots
\end{gathered}
$$

  ```python
  def decimal_to_base(n, k):
      digits = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ" # 用于获得各个d_i

      results = []  # [d_0,...,d_m], 最后再翻转
      while n > 0:
        res = n % k
        results.append(digits[res])
        n //= k

      return ''.join(results[::-1])
  ```

- 10进制小数->k进制小数

  设10进制小数$f = d_{-1} k^{-1} + d_{-2} k^{-2} + \cdots$，则其$k$进制表示为$0.d_{-1} d_{-2} \cdots$，其中$d_{-i} \in \{0, 1, \cdots, k-1\}$。

  可以通过不断乘以k并收集整数部分（向下取整）来依次得到各个$d_{-1}, d_{-2}, \cdots$：

$$\lfloor f*k \rfloor = \lfloor d_{-1} + d_{-2}k^{-1} + \cdots \rfloor = d_{-1},$$
$$f = f*k - d_{-1} = d_{-2}k^{-1} + \cdots$$
$$\lfloor f*k \rfloor = \lfloor d_{-2} + d_{-3}k^{-1} + \cdots \rfloor = d_{-2}$$
$$f = f*k - d_{-2} = d_{-3}k^{-1} + \cdots$$

一般需要设置一下保留多少位小数

```
def decimal_frac_to_base(f, k, precision=10):
  digits = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
  results = []  # d_{-1}, d_{-2}, ...

  for _ in range(precision):
    digit = int(f*k)
    results.append(digits[digit])
    f = f * k - digit
    if f == 0:
      break

  return '0.' + ''.join(results)
```

- k进制整数->10进制整数

  对于k进制数$s = d_m k^m + \cdots + d_1 k^1 + d_0 k^0$，直接按照这个表达式累加各个$d_i k^k$即可
- m进制->n进制：先转为10进制，再转为n进制

质数

- 质数判定：

  设自然数N不是质数，则必然存在一对数字$x \le y$使得如下条件成立：

  $$N = x * y$$
  $$1 < x \le \sqrt{N} \le y < N$$

  因此，只需在$[2, \sqrt{N}]$范围内枚举x，看看是否存在这样的x即可。时间复杂度为$O(\sqrt{N})$

```
from math import sqrt
def judge_prime(n):
  i = 2
  while i <= sqrt(n):
    if n % i == 0:
      return False
    i += 1
  return True
```

- 质数筛选：

对于正整数N，希望求出$1 \sim N$之间所有的质数，使用埃式筛选法。

设标记数组初始化为`vis = [0] * (N+1); vis[0]=1, vis[1]=1`，其中`vis[i]==0`表示它是质数，`vis[i]==1`表示它不是质数。

基于如下观察：若一个数字x是质数，则它的任意倍数$2x, 3x, \cdots$均不是质数。因此，在从前往后遍历$[2, N]$中的每个数时，若发现某个数i在`vis`中的标记为`vis[i]==0`，则说明它不是$[2, i-1]$中任何数字的倍数，则说明它是质数，因此将`vis[i]`保留为0；同时，把$[2, N]$范围内所有i的倍数$2i, 3i, \cdots$的`vis`值全设置为1，因为在i这里发现了一个它的因数。

```
def get_primes(n):
  if n < 2:
    return []

  vis = [0] * (n+1)
  vis[0] = 1
  vis[1] = 1

  primes = []

  for i in range(2, n+1):
    if vis[i] == 0: # i是质数
      primes.append(i)
      j = i * 2
      while j <= n: # 将i的所有倍数标记为1
        vis[j] = 1
        j += i
   return primes
```

- 质因数分解：

  唯一分解定理：任何大于1的正整数都能唯一分解为有限个质数乘积：

  $$N = p_1^{c_1} p_2^{c_2} \cdots p_m^{c_m}$$

  其中幂次c_i均为正整数，p_i均为质数且$p_1 < p_2 < \cdots < p_m$

  质因数分解方法：

  首先求最小的质因数p_1，枚举$[2, \sqrt{N}]$中的所有x，如果N是x的倍数，则x为p_1（如果在这个范围内找不到p_1的话则说明N自己就是质数，质因数分解就是它本身）。然后，令N不断除以p_1，直到其不再能整除p_1,此时已经消除了所有p_1的成分，$N = p_2^{c_2} \cdots p_m^{c_m}$，从$x = p_1$下一位开始继续往后枚举$x$，如果再次出现$N$是$x$倍数的情况，则$x$为$p_2$，依此类推枚举完所有$x$即可得到所有质因数。

```
from math import sqrt
def prime_factors(n):
  factors = []  # p_1,p_2,...
  exps = []   # c_1,c_2,...

  i = 2
  found_factor = False  # 是否在[2,sqrt(n)]中找到了至少一个质因数
```

```
    while i <= sqrt(n):
      if n % i == 0:    # 如果发现一个质因数
        found_factor = True
        factors.append(i)
        exp = 0   # 统计这个质因数的幂次
        while n % i == 0:   # 开始不断消掉这个质因数
          exp += 1
          n /= i
        exps.append(exp)

    if not found_factor:  # n自己是质数
      factors.append(n)
      exps.append(1)

    return factors, exps
```

- 两数互质判定：

  只需求出二者最大公约数，若为1则二者互质

- 裴蜀定理：

  若a,b,x,y,m为整数，则$ax+by=m$有解当且仅当m是$gcd(a,b)$的倍数

应用：

- 判断分数`p/q`在`k`进制下是否为有限小数：充要条件为（通过将`p,q`都除以它们的最大公约数以化为最简分数后）分母`q`对所有质因数都是`k`的质因数（也即分母`q`的质因数是`k`的质因数的子集）

快速幂

求a^x时，若直接把x个a进行乘积效率比较低，可以通过把指数x转为二进制来加快幂运算。

设x的二进制表示为$x=d_0 2^0+d_1 2^1+\cdots+d_k 2^k$（$d_i\in\{0,1\}$），则$a^x=a^{d_0 2^0+d_1 2^1+\cdots+d_k 2^k}=a^{d_0 2^0}a^{d_1 2^1}\cdots a^{d_k 2^k}$。由于$a^{2^i}=(a^{2^{i-1}})^2=(a^{2^{i-2}})^4=\cdots$，因此算出某个$a^{2^j}$后，即可将其进一步用于计算它的2倍、4倍等：$a^{2^{j+1}}=(a^{2^j})^2$，$a^{2^{j+2}}=(a^{2^{j+1}})^2$，...，后边就可以直接在累乘答案上乘以$a^{2^{j+1}}$、$a^{2^j+2}$等，而不需再一个个乘以多个$a$。相比于直接把$x$个$a$相乘的“线性累乘速度”，变为了“每次翻2倍的指数级累乘速度”。这样可以大幅减少所需总计算数量（由$O(x)$简化为$O(\log x)$）。

具体而言，设初始答案为$ans=1$（用于累乘得到最终答案），初始基数$base=a^{2^0}$（$base=a^{2^i}$表示当前可乘到ans上的基数，若其系数$d_i=1$则说明它在x的二进制分解中存在，可以乘到ans上）。对于指数$x=d_0 2^0+d_1 2^1+\cdots+d_k 2^k$（$d_i\in\{0,1\}$），其二进制表示为$d_k\cdots d_1 d_0$，当$x>0$时，每轮循环中通过当前$d_i$是否等于1来决定是否要将当前的$base=a^{2^i}$乘到累乘答案$ans$上，然后每次将$x$右移1位、将基数$base$扩大2倍（由$a^{2^i}\to a^{2^{i+1}}$），从而进行下一轮的观察：

对于第i轮循环，此时x的二进制表示为$x=d_k\cdots d_i$，基数$base=a^{2^i}$

- 若$x\&1=1$，也即当前x的最低位$d_i=1$：

  则说明2^i存在于x的二进制分解表达式中，因此将一个$base$乘到ans上：$ans=ans*base=ans*a^{2^i}$
- 然后对$base$做平方，使得基数扩大2倍变为$base=(a^{2^i})^2=a^{2^{i+1}}$
- 然后将x右移1位，去掉d_i，使得$x=d_k\cdots d_{i+1}$

算法流程：

- 初始化结果`ans=1`，基数`base=a`
- 当`x>0`时，进行如下循环操作：
  - 若`x`的最低位为1（`x & 1 == 1`），则`ans = ans * base`
  - 对`base`做平方：`base = base ** 2`
  - 将`x`右移一位（也即`x//2`）：`x >>= 1`

```
def ppow(a, x):
  ans = 1
  base = a:

  while x > 0:
    if x & 1:
      ans = ans * base
    base = base ** 2
    x >>= 1
  return ans
```

牛顿法

牛顿法用于寻找函数零点$f(x)=0$的解，其从一个初始猜想解x_0开始，不断迭代：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

直到$|x_{n+1}-x_n|<\epsilon$，也即两次迭代间的变化变得非常小。

示例：给定N，求其开根号值$\sqrt{N}$

将其等价转为求$f(x)=x^2-N=0$的根，导数为$f'(x)=2x$，因此迭代式为：

$$x_{n+1}=x_n-\frac{x_n^2-N}{2x_n}=\frac{1}{2}\left(x_n+\frac{N}{x_n}\right)$$

初始值可取$x_0=N$或$x_0=N/2$等

代码实现：

```python
def sqrt_newton(N, eps=1e-10):
    x = N # 初始值
    while True:
        x_next = 0.5 * (x + N / x)
        if abs(x_next - x) < eps:
            return x_next
        x = x_next
```

ACM OJ平台技巧与注意事项

> 用于互联网公司机考。参考卡码网。

处理大量输入

- 处理大量输入（正常来讲可以不断`input()`来接收，但效率会比较低）：使用`sys.stdin.read()`

```python
"""
示例输入：
1
3
1 3 4
1 2 1
2 3 2
"""

import sys

input_ = sys.stdin.read().split('\n')
"""
input_ = ['1', '3', '1 3 4', '1 2 1', '2 3 2']
"""
```

数值精度问题

- 最小值初始化：

  `float('-inf')`

- 大数除法：

  大偶数做除以二操作：使用`//`而不是`/`，来保护精度。

  事实上，对于任何除法操作，只要数学上能够整除，则尽量多使用`//`。

内置函数

排序与字符串操作

- 排序：

  当没有专门考排序时，可以直接使用内置排序函数：

  原地排序：

  ```
  ls = [[1, 2], [3, 4], [3, 5]...]
  ls.sort(key=lambda x: (x[0], x[1]), reverse=False)
  ```

  非原地排序：

  ```
  ls_sorted = sorted(ls, key=lambda x: (x[0], x[1]), reverse=False)
  ```

  可使用`sort`方法中的`key`参数来指定排序标准，例如上例中即为：首先比较每个列表元素的第一项，如果第一项相等再比较第二项。默认从小到大排序，如果想从大到小可以使用`reverse=True`参数。

- 字符串常用操作
  - `split()`：

    默认分隔符是空格：

    ```
    s = '0 1 0'
    s_ls = s.split()
    # ['0', '1', '0']
    ```

    当字符串各个char之间不存在任何分隔符时，只需直接使用`list()`转换即可：

    ```
    s = '010'
    s_ls = list(s)
    # ['0', '1', '0']
    ```

  - 找到第一个出现字符x的位置，若没有该字符则返回-1：`str_.find(x)`
  - 计算字符串中字符x出现的次数：`str_.count(x)`

列表批处理

- 常用内置函数：对列表进行批处理
  - `map()`：

    对一个可迭代对象的所有参数都通过某种函数进行一个变换（例如把每个列表元素都变成整数形式）

    注意第一个参数是变换函数，第二个参数是可迭代对象：

```
ls_char = ['0', '1', '0']
ls_num = list(map(lambda x: int(x), ls_char))
# [0, 1, 0]
```

- `filter()`：

  根据特定过滤条件（一个返回True/False）的函数来从可迭代对象中选择相应的元素

  第一个参数是函数，第二个参数是待处理可迭代对象

```
a = [1,2,3,4,5,6]
b = filter(lambda x:x%2==1, a)
print(list(b))
# [1, 3, 5]
```

- `functools.reduce()`：

  以某种逻辑来将列表中所有元素组合成一个值（例如通过字符串连接、数值累加、数值累乘等方式）

  第一个参数为函数，其输入为每两个相邻元素，输出为这两个相邻元素组合的结果。

  其作用逻辑为从前往后依次合并

  示例：整个数组累乘：

```
from functools import reduce

a = [1,2,3,4,5]
b = reduce(lambda x,y: x*y, a)
b
# 120
"""
逻辑：
[1*2, 3, 4, 5] -> [2, 3, 4, 5]
[2*3, 4, 5] -> [6, 4, 5]
[6*4, 5] -> [24, 5]
24*5 -> 120
""
```

  示例：将列表中的字符串用`"-"`连接起来：

```
from functools import reduce
a = ["orange", "apple", "pear", "banana"]
b = reduce(lambda x,y:x+"-"+y, a)
print(b)
# orange-apple-pear-banana
```

- `collections.Counter`：求出数组中各个元素的数量并保存为字典

```
from collections import Counter
a = [1,1,2,2,2,3,4]
mp = Counter(a)
mp
# Counter({2: 3, 1: 2, 3: 1, 4: 1})
# Counter对象本身就可以起到字典的效果，如：mp[2] = 3, mp[1] = 2。当然也可以使用dict(mp)将其转为真正的字典
```