题目

167. 两数之和 II - 输入有序数组

给定一个已按照升序排列 的有序数组，找到两个数使得它们相加之和等于目标数。
函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

返回的下标值（index1 和 index2）不是从零开始的。
你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。 示例:

输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。

In [1]:

```python
class Solution:
    def twoSum(self, numbers, target):
        left = 0
        right = len(numbers)-1
        while left < right:
            if numbers[left]+numbers[right] == target:
                return[left+1, right+1]
            elif numbers[left]+numbers[right] < target:
                left += 1
            else:
                right -= 1


numbers = [2, 7, 11, 15]
target = 9
s = Solution()
result = s.twoSum(numbers, target)
print(result)
```

```
[1, 2]
```

215. 数组中的第K个最大元素

在未排序的数组中找到第 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。

示例 1:
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

示例 2:
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4

说明: 你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

In [2]:

```python
class Solution:
    def findKthLargest(self, nums, k):
        result = sorted(nums)[-k]
        return result


s = Solution()
nums1 = [3, 2, 1, 5, 6, 4]
k1 = 2
result1 = s.findKthLargest(nums1, k1)
print(result1)
nums2 = [3, 2, 3, 1, 2, 4, 5, 5, 6]
k2 = 4
result2 = s.findKthLargest(nums2, k2)
print(result2)
```

```
5
4
```

347. 前K个高频元素

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

示例 1:
输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]

示例 2: 输入: nums = [1], k = 1
输出: [1]

说明:

你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小

In [3]:

```python
class Solution:
    def topKFrequent(self, nums, k):
        d = {}
        for n in nums:
            d[n] = d.get(n, 0) + 1
        return sorted(d.keys(), key=d.get)[-k:]


s = Solution()
nums1 = [1, 1, 1, 2, 2, 3]
k1 = 2
result1 = s.topKFrequent(nums1, k1)
print(result1)
nums2 = [1]
k2 = 1
result2 = s.topKFrequent(nums2, k2)
print(result2)
```

```
[2, 1]
[1]
```

75. 颜色分类

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，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。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

注意: 不能使用代码库中的排序函数来解决这道题。

示例: 输入: [2,0,2,1,1,0]
输出: [0,0,1,1,2,2]

进阶： 一个直观的解决方案是使用计数排序的两趟扫描算法。
首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。
你能想出一个仅使用常数空间的一趟扫描算法吗？

In [4]:

```python
class Solution:
    def sortColors(self, nums):
        """
        Do not return anything, modify nums in-place instead.
        """
        left = 0
        right = len(nums) - 1
        i = 0
        while i <= right:
            while nums[i] == 2 and i < right:
                nums[i], nums[right] = nums[right], nums[i]
                right -= 1
            while nums[i] == 0 and i > left:
                nums[i], nums[left] = nums[left], nums[i]
                left += 1
            i += 1


s = Solution()
nums = [2, 0, 2, 1, 1, 0]
s.sortColors(nums)
print(nums)
```

```
[0, 0, 1, 1, 2, 2]
```

455. 分发饼干

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。对每个孩子 i ，都有一个胃口值 gi ，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 j ，都有一个尺寸 sj 。如果 sj >= gi ，我们可以将这个饼干 j 分配给孩子 i ，这个孩子会得到满足。你的目标是尽可能满足越多数量的孩子，并输出这个最大数值。

注意:

你可以假设胃口值为正。 一个小朋友最多只能拥有一块饼干。

示例 1:
输入: [1,2,3], [1,1]

输出: 1

解释: 你有三个孩子和两块小饼干，3个孩子的胃口值分别是：1,2,3。
虽然你有两块小饼干，由于他们的尺寸都是1，你只能让胃口值是1的孩子满足。
所以你应该输出1。

In [5]:

```python
class Solution:
    def findContentChildren(self, g, s):
        g.sort()
        s.sort()

        gi = 0  # 口味值
        si = 0  # 尺寸值
        res = 0

        while gi < len(g) and si < len(s):
            if s[si] >= g[gi]:
                si += 1
                gi += 1
                res += 1
            else:
                si += 1
        return res


solution = Solution()
g = [1, 2, 3]
s = [1, 1]
result = solution.findContentChildren(g, s)
print(result)
```

```
1
```

sort 与 sorted 区别

- sort 是应用在 list 上的方法，sorted 可以对所有可迭代的对象进行排序操作。
- list 的 sort 方法返回的是对已经存在的列表进行操作，无返回值，而内建函数 sorted 方法返回的是一个新的 list，而不是在原来的基础上进行的操作。

冒泡排序

冒泡排序（Bubble sort）：时间复杂度O(n^2)
交换排序的一种。其核心思想是：两两比较相邻记录的关键字，如果反序则交换，直到没有反序记录为止。

In [6]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def swap(self, i, j):
        """定义一个交换元素的方法，方便后面调用。"""
        temp = self.r[i]
        self.r[i] = self.r[j]
        self.r[j] = temp

    def bubble_sort_simple(self):
        """
        最简单的交换排序，时间复杂度O(n^2)
        """
        lis = self.r
        length = len(self.r)
        for i in range(length):
            for j in range(i+1, length):
                if lis[i] > lis[j]:
                    self.swap(i, j)

    def bubble_sort(self):
        """
        冒泡排序，时间复杂度O(n^2)
        """
        lis = self.r
        length = len(self.r)
        for i in range(length):
            j = length-2
            while j >= i:
                if lis[j] > lis[j+1]:
                    self.swap(j, j+1)
                j -= 1

    def bubble_sort_advance(self):
        """
        冒泡排序改进算法，时间复杂度O(n^2)
        设置flag，当一轮比较中未发生交换动作，则说明后面的元素其实已经有序排列了。
        对于比较规整的元素集合，可提高一定的排序效率。
        """
        lis = self.r
        length = len(self.r)
        flag = True
        i = 0
        while i < length and flag:
            flag = False
            j = length - 2
            while j >= i:
                if lis[j] > lis[j + 1]:
                    self.swap(j, j + 1)
                    flag = True
                j -= 1
            i += 1

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret

if __name__ == '__main__':
    sqlist = SQList([4,1,7,3,8,5,9,2,6])
    # sqlist.bubble_sort_simple()
    # sqlist.bubble_sort()
    sqlist.bubble_sort_advance()
    print(sqlist)
```

```
 1 2 3 4 5 6 7 8 9
```

简单选择排序

简单选择排序（simple selection sort）:时间复杂度O(n^2)
通过n-i次关键字之间的比较，从n-i+1个记录中选出关键字最小的记录，并和第i（1<=i<=n)个记录进行交换。

In [7]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def swap(self, i, j):
        """定义一个交换元素的方法，方便后面调用。"""
        temp = self.r[i]
        self.r[i] = self.r[j]
        self.r[j] = temp

    def select_sort(self):
        """
        简单选择排序，时间复杂度O(n^2)
        """
        lis = self.r
        length = len(self.r)
        for i in range(length):
            minimum = i
            for j in range(i+1, length):
                if lis[minimum] > lis[j]:
                    minimum = j
            if i != minimum:
                self.swap(i, minimum)

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret


if __name__ == '__main__':
    sqlist = SQList([4, 1, 7, 3, 8, 5, 9, 2, 6, 0])
    sqlist.select_sort()
    print(sqlist)
```

```
 0 1 2 3 4 5 6 7 8 9
```

直接插入排序

直接插入排序（Straight Insertion Sort）:时间复杂度O(n^2)
基本操作是将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增1的有序表。

In [8]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def insert_sort(self):
        lis = self.r
        length = len(self.r)
        # 下标从1开始
        for i in range(1, length):
            if lis[i] < lis[i-1]:
                temp = lis[i]
                j = i-1
                while lis[j] > temp and j >= 0:
                    lis[j+1] = lis[j]
                    j -= 1
                lis[j+1] = temp

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret


if __name__ == '__main__':
    sqlist = SQList([4, 1, 7, 3, 8, 5, 9, 2, 6, 0])
    sqlist.insert_sort()
    print(sqlist)
```

```
 0 1 2 3 4 5 6 7 8 9
```

希尔排序

希尔排序（Shell Sort）是插入排序的改进版本，其核心思想是将原数据集合分割成若干个子序列，然后再对子序列分别进行直接插入排序，使子序列基本有序，最后再对全体记录进行一次直接插入排序。

In [9]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def shell_sort(self):
        """希尔排序"""
        lis = self.r
        length = len(lis)
        increment = len(lis)
        while increment > 1:
            increment = int(increment/3)+1
            for i in range(increment+1, length):
                if lis[i] < lis[i - increment]:
                    temp = lis[i]
                    j = i - increment
                    while j >= 0 and temp < lis[j]:
                        lis[j+increment] = lis[j]
                        j -= increment
                    lis[j+increment] = temp

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret


if __name__ == '__main__':
    sqlist = SQList([4, 1, 7, 3, 8, 5, 9, 2, 6, 0, 123, 22])
    sqlist.shell_sort()
    print(sqlist)
```

```
 0 1 2 3 4 5 6 7 8 9 22 123
```

堆排序

堆是具有下列性质的完全二叉树：
每个分支节点的值都大于或等于其左右孩子的值，称为大顶堆；
每个分支节点的值都小于或等于其做右孩子的值，称为小顶堆；
因此，其根节点一定是所有节点中最大（最小）的值。

堆排序（Heap Sort）就是利用大顶堆或小顶堆的性质进行排序的方法。堆排序的总体时间复杂度为O(nlogn)。

In [10]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def swap(self, i, j):
        """定义一个交换元素的方法，方便后面调用。"""
        temp = self.r[i]
        self.r[i] = self.r[j]
        self.r[j] = temp

    def heap_sort(self):
        length = len(self.r)
        i = int(length/2)
        # 将原始序列构造成一个大顶堆
        # 遍历从中间开始，到0结束，其实这些是堆的分支节点。
        while i >= 0:
            self.heap_adjust(i, length-1)
            i -= 1
        # 逆序遍历整个序列，不断取出根节点的值，完成实际的排序。
        j = length-1
        while j > 0:
            # 将当前根节点，也就是列表最开头，下标为0的值，交换到最后面j处
            self.swap(0, j)
            # 将发生变化的序列重新构造成大顶堆
            self.heap_adjust(0, j-1)
            j -= 1

    def heap_adjust(self, s, m):
        """核心的大顶堆构造方法，维持序列的堆结构。"""
        lis = self.r
        temp = lis[s]
        i = 2*s
        while i <= m:
            if i < m and lis[i] < lis[i+1]:
                i += 1
            if temp >= lis[i]:
                break
            lis[s] = lis[i]
            s = i
            i *= 2
        lis[s] = temp

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret


if __name__ == '__main__':
    sqlist = SQList([4, 1, 7, 3, 8, 5, 9, 2, 6, 0, 123, 22])
    sqlist.heap_sort()
    print(sqlist)
```

```
 0 1 2 3 4 5 6 7 8 9 22 123
```

归并排序

归并排序（Merging Sort）：建立在归并操作上的一种有效的排序算法,该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

In [11]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def swap(self, i, j):
        """定义一个交换元素的方法，方便后面调用。"""
        temp = self.r[i]
        self.r[i] = self.r[j]
        self.r[j] = temp

    def merge_sort(self):
        self.msort(self.r, self.r, 0, len(self.r)-1)

    def msort(self, list_sr, list_tr, s, t):
        temp = [None for i in range(0, len(list_sr))]
        if s == t:
            list_tr[s] = list_sr[s]
        else:
            m = int((s+t)/2)
            self.msort(list_sr, temp, s,  m)
            self.msort(list_sr, temp, m+1, t)
            self.merge(temp, list_tr, s, m, t)

    def merge(self, list_sr, list_tr, i, m,  n):
        j = m+1
        k = i
        while i <= m and j <= n:
            if list_sr[i] < list_sr[j]:
                list_tr[k] = list_sr[i]
                i += 1
            else:
                list_tr[k] = list_sr[j]
                j += 1

            k += 1
        if i <= m:
            for l in range(0, m-i+1):
                list_tr[k+l] = list_sr[i+l]
        if j <= n:
            for l in range(0, n-j+1):
                list_tr[k+l] = list_sr[j+l]

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret

if __name__ == '__main__':
    sqlist = SQList([4, 1, 7, 3, 8, 5, 9, 2, 6, 0, 12, 77, 34, 23])
    sqlist.merge_sort()
    print(sqlist)
```

```
 0 1 2 3 4 5 6 7 8 9 12 23 34 77
```

快速排序

快速排序（Quick Sort）由图灵奖获得者Tony Hoare发明，被列为20世纪十大算法之一。冒泡排序的升级版，交换排序的一种。快速排序的时间复杂度为O(nlog(n))。

In [12]:

```python
class SQList:
    def __init__(self, lis=None):
        self.r = lis

    def swap(self, i, j):
        """定义一个交换元素的方法，方便后面调用。"""
        temp = self.r[i]
        self.r[i] = self.r[j]
        self.r[j] = temp

    def quick_sort(self):
        """调用入口"""
        self.qsort(0, len(self.r)-1)

    def qsort(self, low, high):
        """递归调用"""
        if low < high:
            pivot = self.partition(low, high)
            self.qsort(low, pivot-1)
            self.qsort(pivot+1, high)

    def partition(self, low, high):
        """
        快速排序的核心代码。
        其实就是将选取的pivot_key不断交换，将比它小的换到左边，将比它大的换到右边。
        它自己也在交换中不断变换自己的位置，直到完成所有的交换为止。
        但在函数调用的过程中，pivot_key的值始终不变。
        :param low:左边界下标
        :param high:右边界下标
        :return:分完左右区后pivot_key所在位置的下标
        """
        lis = self.r
        pivot_key = lis[low]
        while low < high:
            while low < high and lis[high] >= pivot_key:
                high -= 1
            self.swap(low, high)
            while low < high and lis[low] <= pivot_key:
                low += 1
            self.swap(low, high)
        return low

    def __str__(self):
        ret = ""
        for i in self.r:
            ret += " %s" % i
        return ret

if __name__ == '__main__':
    sqlist = SQList([4, 1, 7, 3, 8, 5, 9, 2, 6, 0, 123, 22])
    sqlist.quick_sort()
    print(sqlist)
```

```
 0 1 2 3 4 5 6 7 8 9 22 123
```