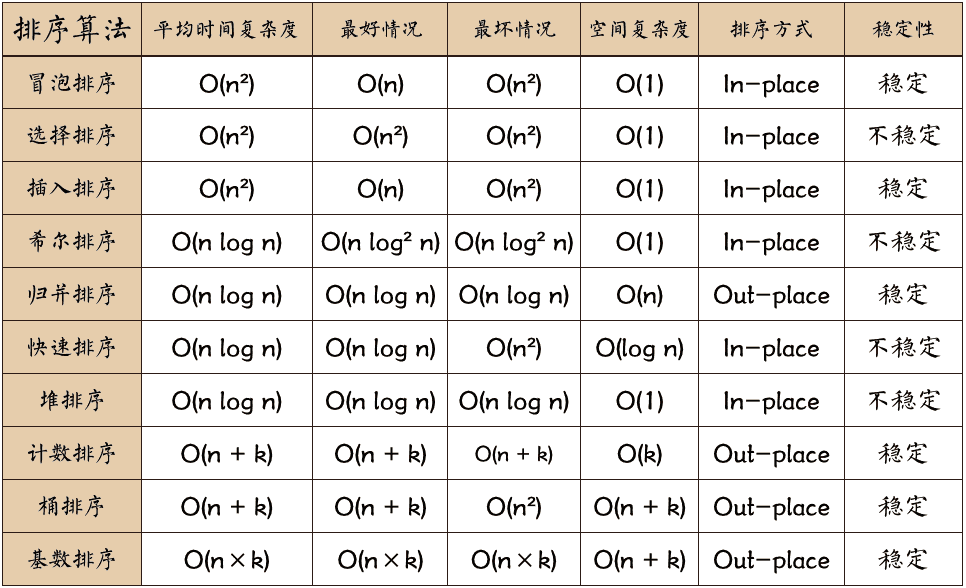
首先放一张排序算法复杂度的对比图



以下程序均默认为升序排列，排序方式in-place表示是否需要额外数组空间保存临时结果。排序的稳定性则表示，如果具有相等键的若干个对象在同一个序列中，那么排好序的有序序列这些相等键的对象会保留排序之前的相对顺序不变。例如1，2，3，2排好序之后为1，2（第二个），2（第一个），3则说明算法不稳定。

**冒泡排序**

冒泡排序名字由来是因为越小的元素会在交换之后慢慢浮到数列的顶端，就如同气泡会在水中上浮到顶端一样。算法流程是重复遍历要排序的数组，如果相邻的两个元素位置相反就要交换，例如升序排序需要交换逆序对，将小的放到前面。

算法原理：1.比较相邻元素，如果第一个大于第二个则交换；2.从开始的一对到最后一对均做同样的操作，每次遍历完成后，最大的数字应该会到数组的最后一位；3.针对除了已经定位的元素重复上述操作（定位指的是已经确定了位置的，第i次遍历可以确定数组最后i位数字的位置）。

def bubble\_sort(self, nums):

n = len(nums)

for i in range(n-1):

for j in range(n-1-i):

if nums[j] > nums[j+1]:

nums[j], nums[j+1] = nums[j+1], nums[j]

return nums

**选择排序**

选择排序的方式为遍历数组，找出当前趟的最小的一个元素，和当前趟的第一个元素交换。当前趟指的是第i趟遍历时从第i-1位元素开始。最终得到一个有序序列。每次选择一个最符合条件的元素放到正确位置，所以叫选择排序。

算法原理：1.遍历数组找出最小的元素位置，和第一个元素交换；2.从第2个元素开始继续上述操作，每次遍历完成后会得到前i个有序元素排列；3.重复操作直到数组有序。

def select\_sort(self, nums):

n = len(nums)

for i in range(n-1):

index = i

for j in range(i+1, n):

if nums[j] < nums[index]:

index = j

nums[i], nums[index] = nums[index], nums[i]

return nums

**插入排序**

插入排序的基本思想是将一个元素插入到已经排好序的有序序列中，从而得到一个长度+1的有序序列。在实现方面外层循环记录需要插入的元素，内层循环寻找需要插入的位置，并且将待插入的元素后面的元素统一向后移位。现实中的类似用例为打扑克时需要洗牌，将扑克牌按照点数从小到大排列，在拿手牌的过程中逐个插入有序序列中。

算法原理：1.确定要插入到有序序列中的元素，通过外层循环实现，范围是[1, n-1]；2.内层循环寻找第一个小于该元素的元素位置，则该位置的后一位就是待插入元素的正确位置；3.重复操作直到序列有序。

def insert\_sort(self, nums):

n = len(nums)

for i in range(1, n):

j = i - 1

key = nums[i]

while j >= 0 and nums[j] > key:

nums[j+1] = nums[j]

j -= 1

nums[j+1] = key

return nums

**希尔排序**

希尔排序是插入排序的一种改进版本，又称缩小增量排序。插入排序算法选择一个元素之后每次移动一步并进行复制，导致排序过程中一个元素可能会经过多次复制和移动。希尔排序则将元素按照下标的一定增量对元素进行分组，对每组使用插入排序；并逐次减少增量，从而每组包含的关键词越来越多，但每次插入排序之后该分组的元素会更接近最终有序的结果，相当于每次向最终结果靠近一点。当下标的增量减为1时，整个序列有序。

希尔排序的改进基础：1.插入排序对几乎排好序的序列操作时效率非常高，接近线性；2.插排一般来说低效，因为每次只能将数据移动一位。基于上述点，将数据分组排序，组内有序在对组内合并时可以做到几乎有序并且组内排序时可以做到数据移动多位，加快了移动效率。

算法原理：1.取一个小于n的整数作为第一个增量d，一般为n/2，将全部元素按照下标增量分组，在组内进行插入排序；2.取第二个增量d2=d/2，继续分组做插排，直到增量减为0排序完成。

def shell\_sort(self, nums):

n = len(nums)

gap = n // 2

while gap > 0:

for i in range(gap, n):

key = nums[i]

j = i - gap

while j >= 0 and nums[j] > key:

nums[j+gap] = nums[j]

j -= gap

nums[j+gap] = key

gap = gap // 2

return nums

**归并排序**

归并排序是使用分治法解决排序问题的，具体来说，递归的将数组分割成更小的数组，例如一分为二，然后对分割后的数组进行进一步的分割，或达到最小可排序的数组长度后结束。而对于合并阶段，声明一个临时数组保存排序后的结果，建立待合并数组遍历指针，每次取出其中的较小值放入临时数组中，直到一方遍历完。其后将剩余有数字的数组的剩余所有元素拷贝到临时数组中。最后将临时数组中对应排序完成的部分赋值为原数组。

算法原理：1.如果当前数组长度需要排序（即长度大于1），将数组分为两部分并分别递归排序，否则直接返回结束当前递归；2.对划分后的数组排序完成后对数组进行合并；3.合并操作为遍历两个待合并数组，每次取出最小值赋给临时数组，并在一个数组遍历完成后将另一个数组的剩余元素赋值给临时数组，最终将临时数组拷贝到原数组的对应位置处。

def merge\_sort(self, nums):

def merge(nums, l, mid, r):

i = l

j = mid + 1

k = l

while i <= mid and j <= r:

if nums[i] > nums[j]:

tmp[k] = nums[j]

j += 1

else:

tmp[k] = nums[i]

i += 1

k += 1

while i <= mid:

tmp[k] = nums[i]

k += 1

i += 1

while j <= r:

tmp[k] = nums[j]

k += 1

j += 1

nums[l:r+1] = tmp[l:r+1]

def \_merge\_sort(nums, l, r):

if l < r:

mid = (l + r) // 2

\_merge\_sort(nums, l, mid)

\_merge\_sort(nums, mid+1, r)

merge(nums, l, mid, r)

tmp = [0] \* len(nums)

\_merge\_sort(nums, 0, len(nums)-1)

return nums

**快速排序**

快排同样也是利用分治思想解决排序问题，但与归并不同的是，归并是通过构造临时数组保存排好序的结果，并且是在子数组保证有序之后进行合并，而快排则是在数组本身空间中修改，在选定一个元素后，将数组修改为所有小于该元素的元素在该元素之前，而所有大于该元素的元素在该元素之后的状态。在对当前数组做完如此操作之后，这三个部分整体保持有序，然后再对前后两部分继续做如上操作直到数组长度为1.

算法原理：1.对数组进行划分，选出一个元素作为比较的基准（pivot），排列该数组使得所有小于基准的在基准之前，大于基准的在基准之后，这样得到的数组基准处于数组的中位数位置。称为分区操作（partition）；2.递归的对分区后的小于基准和大于基准的子数组进行上述排序操作。

def quick\_sort(self, nums):

def \_partition(nums, l, r):

pivot = nums[l]

start = l

end = r

while start < end:

while start < end and nums[end] > pivot:

end -= 1

if start < end:

nums[start] = nums[end]

while start < end and nums[start] <= pivot:

start += 1

if start < end:

nums[end] = nums[start]

nums[start] = pivot

return start

def \_quick\_sort(nums, l, r):

if l < r:

mid = \_partition(nums, l, r)

\_quick\_sort(nums, l, mid-1)

\_quick\_sort(nums, mid+1, r)

\_quick\_sort(nums, 0, len(nums)-1)

return nums

**堆排序**

堆排序指的是利用二叉堆进行数组的排序。二叉堆是完全二叉树，即在构造过程中会尽量保持子节点是由左向右填充的。堆的每个节点的值都大于等于其子节点的值（最大堆，最小堆相反），这里以最大堆为例。

堆的存储可以使用数组模拟，下标为i的节点的父节点下标为(i-1)//2，其左右子节点分别为(2\*i+1)和(2\*i+2)，因为坐标是从0开始的。数组对应的二叉堆的形状可以考虑成将数组元素逐个填充到一个完全二叉树中。

堆排序有如下操作，保证最大值始终位于根节点：1.最大堆调整（max\_heapify），将堆的末端子节点调整为小于夫节点的状态。2.创建最大堆（bulid\_heap）：将堆中的数据重新排序，使其满足最大堆的性质，通常是加入一个节点后需要调用一次max\_heapify；3.堆排序（heap\_sort）：每次移除在根节点的数据并记录，并将当前最右的数据移动到根节点重新调整使其保持最大堆性质。

堆排序类似于选择排序，每次从中选择一个最大值放到数组首，但是利用树结构存储子节点小于父节点的性质，在选择最大值后的调整只需要O(lgn)，而选择排序则需要O(n).

算法原理：1.使用数组模拟最大堆的结构；2.对所有非叶子节点进行调整，使其满足最大堆的性质，而非叶子节点的范围是[0,(n-1)//2]；3.取出根节点即最大值并记录，然后将最右边的元素放入根节点中，并进行调整；4.重复步骤3直到得到有序序列，即所有节点都已经取出来。

def heap\_sort(self, nums):

def max\_heapify(start, end):

# 指向当前节点和子节点

father = start

son = 2 \* start + 1

# 保证子节点有效，即没有超过数组范围

while son <= end:

# 选择左右子节点中最大的值代替父节点位置

if son + 1 <= end and nums[son] < nums[son+1]:

son += 1

# 已经满足最大堆性质返回

if nums[father] > nums[son]:

return

# 调整节点满足最大堆，并继续向下调整

nums[father], nums[son] = nums[son], nums[father]

father = son

son = 2 \* father + 1

n = len(nums)

# 建立最大堆的过程，对所有非叶子节点进行调整，并且从下向上遍历，保证对于某一非叶子节点，其子节点均满足最大堆性质

for i in range((n-1)//2, -1, -1):

max\_heapify(i, n-1)

# 每次将根节点（最大值）换到当前数组最后一位，并将最右元素放入根节点进行调整，保证根节点为最大值

for i in range(n-1, 0, -1):

nums[0], nums[i] = nums[i], nums[0]

max\_heapify(0, i-1)

return nums

**计数排序**

计数排序是一种稳定的线性时间排序算法。和名字一样，需要对所有数字出现的次数进行计数，得到所有数字的出现频数之后倒序访问，按照出现频次依次填充到建立的结果数组中即可。因为要对所有数字计数，所以要求数字的出现范围很小，如果出现1-10001000范围的数字则计数排序无法运行，因为这个空间开不下。

算法原理：1.对数组内所有数字出现的次数计数，得到所有数字的出现频数；2.频数数组累加，得到每个数字对应出现在结果数组中的位置+1（如果从0开始索引）;3.遍历原数组对所有数字放到对应的频数数组标记的位置上

def count\_sort(self, nums):

min\_value, max\_value = nums[0], nums[0]

ans = [-1] \* len(nums)

for i in range(1, len(nums)):

if min\_value > nums[i]:

min\_value = nums[i]

elif max\_value < nums[i]:

max\_value = nums[i]

count = [0] \* (max\_value - min\_value + 1)

for i in nums:

count[i-min\_value] += 1

for i in range(1, len(count)):

count[i] += count[i-1]

for num in nums:

count[num-min\_value] -= 1

ans[count[num-min\_value]] = num

return ans

**桶排序**

桶排序的算法同样采用分治方法，首先构造若干个桶用于存放数组中的部分元素，将数组按照特定方式映射到对应桶中，映射之后的桶之间的元素保证按照桶的顺序是整体有序的，即范围有序。然后对各个桶进行排序得到单桶有序。最后按顺序合并所有桶内数据即得到有序序列。

算法原理：1.按照最大值和最小值生成对应数量的桶，根据数据范围调整，例如可以每个桶存10个数的范围，即max\_value//10~min\_value//10；2.遍历数组将元素放入到对应桶中，映射函数要和1保持一致；3.对每个桶进行排序，得到桶内有序序列；4.按顺序合并所有桶元素，得到有序序列。

def bucket\_sort(self, nums):

# 得到最大最小值确定桶的个数

max\_value, min\_value = max(nums), min(nums)

buckets = [[] for \_ in range(max\_value//10 - min\_value//10 + 1)]

# 根据划分桶的方式映射数字并存入桶

for num in nums:

index = num//10 - min\_value//10

buckets[index].append(num)

nums.clear()

# 按顺序合并，内部采用插入排序

for l in buckets:

nums.extend(self.insert\_sort(l))

return nums

**基数排序**

基数排序是利用数字的基数信息得到对应的键值，将数字根据键值放入到对应的桶中，最终有序合并桶得到有序序列的算法，属于稳定性算法。分为LSD（最低位优先）和MSD（最高位优先），分别表示放入桶中的顺序应该从数字低位还是高位开始。直到所有位的数字都经历过一次排序，将所有桶依次连接起来得到一个有序序列。

算法原理：1.首先求出最大数字的位数，用于确定循环次数；2.根据基数构造对应的桶，通常基数为10；3.从低位开始，取得每个数字的低位数字作为键值，将数字放入桶中；4.依次合并桶中的数组；5.重复34直到所有位数遍历完成。

def radix\_sort(self, nums, radix=10):

import math

# 求出数组中最大数的位数，用于确定排序循环次数

max\_digit = math.ceil(math.log(max(nums), radix))

# 构造存储数字的桶

bucket = [[] for \_ in range(radix)]

for i in range(max\_digit):

# 将数字根据规则放入对应桶中，这里是取最低位最高位的顺序

for num in nums:

bucket[num % (radix \*\* (i+1)) // (radix \*\* i)].append(num)

nums.clear()

# 按顺序合并

for l in bucket:

nums.extend(l)

bucket = [[] for \_ in range(radix)]

return nums