GitHub：https://github.com/Max-Liuhu/keeplearning

排序篇

- 直接插入排序

二路归并排序

- 介绍
  - 归并排序是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。归并排序是一种稳定的排序方法。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2-路归并。
- 算法思路
  1. 把长度为n的输入序列分成两个长度为n/2的子序列；
  2. 对这两个子序列分别采用归并排序；
  3. 将两个排序好的子序列合并成一个最终的排序序列。
- 复杂度与稳定性
  - 空间复杂度：
  - 时间复杂度：
  - 最佳情况：最佳情况：T(n) = O(n)
  - 最差情况：T(n) = O(nlogn)
  - 平均情况：T(n) = O(nlogn)
  - 属于稳定性排序
- 代码

```
def merge_array(L, first, mid, last, temp):
    """
    合并的函数，合并数组
    # 将序列L[first...mid]与序列L[mid+1...last]进行合并
    """
    # 对i,j,k分别进行赋值
    i, j, k = first, mid + 1, 0
    # 当左右两边都有数时进行比较，取较小的数
    while i <= mid and j <= last:
        if L[i] <= L[j]:
            temp[k] = L[i]
            i += 1
        else:
            temp[k] = L[j]
            j += 1
        k += 1

    # 以下两个while只会执行两个
    # 如果左边序列还有数
    while i &lt;= mid:
        temp[k] = L[i]
        i += 1
        k += 1

    # 如果右边序列还有数
    while j &lt;= last:
        temp[k] = L[j]
        j += 1
        k += 1

    # 将temp当中该段有序元素赋值给L待排序列使之部分有序
    for x in range(0, k):
        L[first + x] = temp[x]


def divide_array(L, first, last, temp):
    """分组"""
    if first < last:
        mid = (int)((first + last) / 2)
        # 使左边序列有序
        divide_array(L, first, mid, temp)
        # 使右边序列有序
        divide_array(L, mid + 1, last, temp)
        # 将两个有序序列合并
        merge_array(L, first, mid, last, temp)


# 归并排序的函数
def merge_sort(L):
    # 声明一个长度为len(L)的空列表
    temp = len(L) * [None]
    # 调用归并排序
    divide_array(L, 0, len(L) - 1, temp)
    return L
```

基数排序

- 介绍
  - 基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为O(kn),为数组长度，k为数组中的数的最大的位数
- 算法思路
  1. 取得数组中的最大数，并取得位数；
  2. arr为原始数组，从最低位开始取每个位组成radix数组；
  3. 对radix进行计数排序（利用计数排序适用于小范围数的特点）；

- 复杂度与稳定性

    - 空间复杂度：O(k),辅助空间需要k个队列
    - 时间复杂度：
    - 最佳情况：最佳情况：T(n) = O(n * k)
    - 最差情况：T(n) = O(n * k)
    - 平均情况：T(n) = O(n * k)
    - 属于稳定性排序
        - 队列，先进先出

- 代码

```
def radix_sort(lst):
    """基数排序"""
    if not lst:
        return lst


   RADIX = 10
   placement = 1

   # 获取最大值，最后为循环退出出口
   max_digit = max(lst)
   while placement &lt; max_digit:
       # 0到9的柱状桶，使用列表表示，用于存放基数相同的数据
       buckets = [list() for _ in range(RADIX)]
       # 遍历列表，按基数分类放入桶中
       for i in lst:
           tmp = (i // placement) % RADIX
           buckets[tmp].append(i)

       # 从低到高，将桶中数据依次放入到列表中
       a = 0
       # 相同的数据，在原列表中靠前，分配到桶中时也靠前，取出时也先取出，b也从0开始，所以是稳定性排序
       for b in range(RADIX):
           buck = buckets[b]
           for i in buck:
               lst[a] = i
               a += 1

       # 进入下一个循环
       placement *= RADIX
   return lst</code></pre></li>
```

交换排序

交换排序之冒泡排序

- 算法思路

    i. 将序列当中的左右元素，依次比较，保证右边的元素始终大于左边的元素；（ 第一轮结束后，序列最后一个元素一定是当前序列的最大值）
    ii. 对序列当中剩下的n-1个元素再次执行步骤1。
    iii. 对于长度为n的序列，一共需要执行n-1轮比较

- 复杂度与稳定性

    - 空间复杂度：
    - 时间复杂度：
    - 最佳情况：T(n) = O(n) 当输入的数据已经是正序时
    - 最差情况：T(n) = O(n2) 当输入的数据是反序时
    - 平均情况：T(n) = O(n2)
    - 属于稳定性排序

- 代码

```
def bubble_sort(collection):
    """冒泡排序"""
    length = len(collection)
    for i in range(length - 1):
        swapped = False
        for j in range(length - 1 - i):
            if collection[j] > collection[j + 1]:
                swapped = True
                collection[j], collection[j + 1] = collection[j + 1], collection[j]
        # 当某趟遍历时未发生交换，则已排列完成，跳出循环
        if not swapped:
            break
    return collection
```

- 代码优化方向之一：设置一标志性变量pos,用于记录每趟排序中最后一次进行交换的位置。由于pos位置之后的记录均已交换到位,故在进行下一趟排序时只要扫描到pos位置即可。

```
def bubble_sort(collection):
  """冒泡排序优化之记录每一趟最后交换的位置"""
  length = len(collection)
  i = length-1
  # pos用于记录最后一次交换的位置，并将其赋值给i
  while i > 0:
      pos = 0
      for j in range(i):
          if collection[j] > collection[j + 1]:
              pos = j
              collection[j], collection[j + 1] = collection[j + 1], collection[j]
      i = pos
  return collection
```

- 代码优化方向之二：传统冒泡排序中每一趟排序操作只能找到一个最大值或最小值,我们考虑利用在每趟排序中进行正向和反向两遍冒泡的方法一次可以得到两个最终值(最大者和最小者) , 从而使排序趟数几乎减少了一半

```
def bubble_sort(collection):
    """冒泡排序优化之正反两方向冒泡"""
    # 标记反向
    low =0
    # 标记正向比较结束位置
    high = len(collection)-1
    while low < high:
        # 正向找最大
        for j in range(low,high):
            if collection[j] > collection[j + 1]:
                collection[j], collection[j + 1] = collection[j + 1], collection[j]
        high -= 1

        # 反向找最小
        for j in range(high,low,-1):
            if collection[j] > collection[j - 1]:
                collection[j], collection[j - 1] = collection[j - 1], collection[j]
        low += 1
    return collection
```

交换排序之快速排序

- 算法思路：快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）

i. 从数列中挑出一个元素，称为 "基准"（pivot）；
ii. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
iii. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序

- 复杂度与稳定性
    - 空间复杂度：O(nlog2n)
    - 时间复杂度：
    - 最佳情况：T(n) = O(nlog2n)
    - 最差情况：T(n) = O(n2)
    - 平均情况：T(n) = O(nlog2n)
    - 属于不稳定性排序
- 优缺点及应用场景
    - 快速排序的运行时间与划分是否对称有关
    - 当递归到规模较小的时候采用直接插入排序法
    - 不对称划分及其造成运行时间的增加，中枢元素的选取，从头尾中间最后取三位中的不大不小那个
    - 一大特点:所有内部排序中平均性能最优的排序算法
- 代码

```
    def quick_sort_3partition(sorting, left, right):
        """
        :param sorting: list
        :param left: int
        :param right: int
        :return:
        """
        # 递归出口
        if right <= left:
            return
        # i为sorting中从左到右依次比较的元素索引，初始化为列表的起始位置
        a = i = left
        b = right
        pivot = sorting[left]
        while i <= b:
            # 比基准值小则放到左边
            if sorting[i] < pivot:
                sorting[a], sorting[i] = sorting[i], sorting[a]
                a += 1
                i += 1
            # 比基准值大则放到右边
            elif sorting[i] > pivot:
                sorting[b], sorting[i] = sorting[i], sorting[b]
                b -= 1
            # 相等则比较下一个值
            else:
                i += 1
        quick_sort_3partition(sorting, left, a - 1)
        quick_sort_3partition(sorting, b + 1, right)



# 优化方向
# 1. 中枢选取优化
# 2. 子序列小规模时改为直接插入排序
```

选择排序

选择排序之简单选择排序

- 算法思路：

i. 初始状态：无序区为R[1..n]，有序区为空；
ii. 第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序 从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和

R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
iii. n-1趟结束，数组有序化了

- 主要的两层循环

  - 第一层循环：依次遍历序列当中的每一个元素
  - 第二层循环：将遍历得到的当前元素依次与余下的元素进行比较，符合最小元素的条件，则交换

- 复杂度与稳定性

  - 空间复杂度：

  - 时间复杂度：

  - 最佳情况：T(n) = O(n2)

  - 最差情况：T(n) = O(n2)

  - 平均情况：T(n) = O(n2)

  - 属于不稳定性排序：

  - 比如：[2,2,1]

  - 优缺点及应用场景

  - 注意与直接插入排序比较，当初始表基本有序时选择直接插入排序，其时间复杂度O(n)

- 代码

```
# 简单选择排序的关键点在于：从未排序的序列中选择出最小的放在未排序序列的最前端
def select_sort(nums):
    '''简单选择排序'''
    # 依次遍历序列中的每一个元素，n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果
    for x in range(0, len(nums) - 1):
        # 将当前位置的元素定义此轮循环当中的最小值
        minimum = nums[x]

        # 将该元素与剩下的元素依次比较寻找最小元素
        for i in range(x + 1, len(nums)):
            # 当小时才交换，则是稳定排序
            if nums[i] < minimum:
              # 每一次比较，当发现比当前临时最小值还要小时，及时交换更新
                nums[i], minimum = minimum, nums[i]
        # 遍历比较后，将比较后得到的真正的最小值赋值给当前位置
        nums[x] = minimum

    return nums
```

选择排序之堆排序

- 算法思路：

  i. 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；
  ii. 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；
  iii. 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后 再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元 素个数为n-1，则整个排序过程完成。

- 复杂度与稳定性

  - 空间复杂度：O(1)
  - 时间复杂度：
  - 最佳情况：T(n) = O(nlogn)
  - 最差情况：T(n) = O(nlogn)
  - 平均情况：T(n) = O(nlogn)
  - 属于不稳定性排序：
  - 例如L=[1,2,2]

- 优缺点及应用场景

  - 堆排序需要的辅助空间小于快速排序，不会出现快速排序可能出现的最坏情况

- 代码

```
def heapify(unsorted, index, heap_size):
    '''构建大根堆'''
    # 定义当前节点为临时最大值的索引largest
    largest = index
    # 获取当前节点下标(index)的左右子节点下标
    left_index = 2 * index + 1
    right_index = 2 * index + 2

   # 在不超过堆容量时，进行左节点与父节点的比较，若比父节点大则更新最大值的下标
   if left_index &lt; heap_size and unsorted[left_index] &gt; unsorted[largest]:
       largest = left_index

   # 在不超过堆容量时，进行右节点与父节点的比较，若比父节点大则更新最大值的下标
   if right_index &lt; heap_size and unsorted[right_index] &gt; unsorted[largest]:
       largest = right_index

   # 若当前节点并不是最大值，则进行交换调整，同时此处也是递归出口
   if largest != index:
       # 将当前节点的值与最大值进行互换，即父节点与子节点数据交换
       unsorted[largest], unsorted[index] = unsorted[index], unsorted[largest]
```

```
        # 针对交换后的孩子节点的左右孩子节点进行递归调整判断
        # 这一步必须要
        heapify(unsorted, largest, heap_size)


def heap_sort(unsorted):
    '''堆排序'''
    n = len(unsorted)
    # 从n // 2 - 1节点处依次构建堆，为最后一个叶子节点的父节点
    for i in range(n // 2 - 1, -1, -1):
      # 总共有n//2次遍历，每次遍历都有一个递归函数
        heapify(unsorted, i, n)

    # 执行循环：
    # 1. 每次取出堆顶元素置于序列的最后(len - 1, len - 2, len - 3...)
    # 2. 调整堆，使其继续满足大顶堆的性质，注意实时修改堆中序列的长度
    for i in range(n - 1, 0, -1):
        # i为序列下标从后往前移动，所以最大值为于列表的最后
        # 将抽象数据结构堆的最大值即索引0位置的数值移动i位置
        unsorted[0], unsorted[i] = unsorted[i], unsorted[0]
        heapify(unsorted, 0, i)
    return unsorted</code></pre></li>
```

插入排序

插入排序之简单插入排序

- 算法思路：

    i. 从第一个元素开始，该元素可以认为已经被排序；
    ii. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
    iii. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
    iv. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
    v. 将新元素插入到该位置后；
    vi. 重复步骤2~5。

- 主要的两层循环：

    - 第一层循环：遍历待比较的所有数组元素（从第二个元素开始）
    - 第二层循环：将上层循环选择的元素（selected）与已经排好序的所有元素（ordered）相比较，从选择元素的前面一个开始直到数组的起始位置。如果selected < ordered，那么将二者交换位置，继续遍历；反之，留在原地，选择下一个元素

- 复杂度与稳定性

    - 空间复杂度：O(1)
    - 时间复杂度：
    - 最佳情况：输入数组按升序排列。T(n) = O(n)
    - 最坏情况：输入数组按降序排列。T(n) = O(n2)
    - 平均情况：T(n) = O(n2)
    - 属于稳定性排序

- 优缺点及应用场景

- 代码

```
def insert_sort(nums):
    '''直接插入排序'''
    # 遍历数组中的所有元素，其中0号索引元素默认已排序，因此从1开始
    # 当nums元素数量为空或者1时，不会进入for循环
    # i为待插入元素的索引位置，此次遍历将确认nums[i]应该插入的位置
    for i in range(1, len(nums)):
        # 将该元素与已排序好的前序数组依次比较，如果该元素小，则交换
        # range(x,0,-1):从x倒序循环到1，依次比较，
        # 每次比较如果小于会交换位置，正好按递减的顺序
        # 每次内层循环开始时，j为待插入元素位置，最终循环结束时，j即为插入位置
        for j in range(i, 0, -1): #这里j时取不到0的
            # 判断：如果符合条件则交换,并由此处可见直接插入排序为稳定性排序
            if nums[j] < nums[j - 1]:
              # 交换需要辅助空间，空间复杂度O(1)
                nums[j], nums[j - 1] = nums[j - 1], nums[j]
            else:
                break
    return nums
```

插入排序之折半插入排序

- 在直接插入排序中，查找插入位置时使用二分查找的方式

插入排序之希尔排序

- 学习基础：希尔排序是基于简单插入排序的

- 算法思路：

- 复杂度与稳定性

    - 空间复杂度：O(1)
    - 时间复杂度：
    - 最佳情况：
    - 最坏情况：当n在特定范围时，为O(n2)
    - 平均情况：当n在特定范围时，为O(n1.3)

    - 属于不稳定性排序

- 优缺点及应用场景

    - 适用性：只适用于当线性表为顺序存储的情况

- 代码

```
def shell_sort(nums):
    # 初始化gap值，此处利用序列长度的一半为其赋值
    gap = len(nums) // 2
    # 第一层循环：依次改变gap值对列表进行分组,当gap等于1时进行最后一遍循环
    while gap >= 1:
        # 下面：利用直接插入排序的思想对分组数据进行排序
        # range(gap, len(L)):由于数组下标从0开始，所以从gap开始依次比较
        # gap分组比较，同等的数值可能分部不同组而导致后一组比较后交换，而前一组比较后不交换
        # 故而破坏稳定性，成为不稳定性排序
        # 这两个for循环则是一个小型的直接插入排序
        for i in range(gap, len(nums)):
            # range(i, 0, -gap):从i开始与选定元素开始倒序比较
            # 每个比较元素之间间隔gap
            for j in range(i, 0, -gap):
                # 如果该组当中两个元素满足交换条件，则进行交换
                if nums[j] < nums[j-gap]:
                    nums[j],nums[j-gap] = nums[j-gap],nums[j]
                else:
                    break
        gap = gap // 2
    return nums

# 自定义gap
def shell_Sort(nums):
    lens = len(nums)
    gap = 1
    # 动态定义间隔序列
    while gap < lens // 3:
        gap = gap * 3 + 1
    while gap > 0:
        for i in range(gap, lens):
            curNum, preIndex = nums[i], i - gap  # curNum 保存当前待插入的数
            while preIndex >= 0 and curNum < nums[preIndex]:
                nums[preIndex + gap] = nums[preIndex] # 将比 curNum 大的元素向后移动
                preIndex -= gap
            nums[preIndex + gap] = curNum  # 待插入的数的正确位置
        gap //= 3  # 下一个动态间隔
    return nums
```

对比记忆

1. 简单选择排序与直接插入排序的区别，应用选择，稳定性

选取排序算法需要考虑的因素：

1. 待排序的元素数量n；

2. 元素本身信息量的大小；

3. 当n较少时(n《 50)，直接插入排序或者简单选择排序，由于插入排序移动较多，当记录本身信息很多时，使用简单选择排序；