性能比较

排序算法	时间复杂度（平均）	时间复杂度（最差）	时间复杂度（最好）	空间复杂度	稳定性
快速排序	O($nlog_2n$)	O(n^2)	O($nlog_2n$)	O($nlog_2n$)	不稳定
冒泡排序	O(n^2)	O(n^2)	O(n)	O(1)	稳定
选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定
归并排序	O($nlog_2n$)	O($nlog_2n$)	O($nlog_2n$)	O(n)	稳定
插入排序	O(n^2)	O(n^2)	O(n)	O(1)	稳定
希尔排序	O($n^{1.3}$)	O(n^2)	O(n)	O(1)	不稳定
堆排序	O($nlog_2n$)	O($nlog_2n$)	O($nlog_2n$)	O(1)	不稳定
桶排序	O($n+k$)	O(n^2)	O(n)	O($n+k$)	稳定
计数排序	O($n+k$)	O($n+k$)	O($n+k$)	O($n+k$)	稳定
基数排序	O($n*k$)	O($n*k$)	O($n*k$)	O($n+k$)	稳定

快速排序

原理：对于任意一个需要排序的序列，首先选择序列中的人一个数为关键数据，然后将所有比它小的数都放左边，所有比它大的数都放右边，这就完成了一次快速排序，如此递归完成所有排序。

```
def quick_sort(data):
    """
    快速排序
    """
    if len(data) <= 1:
        return data
    else:
        base = data[0]
        left, right = [], []
        data.remove(base)
```

```
        for num in data:
            if num >= base:
                right.append(num)
            else:
                left.append(num)
        return quick_sort(left) + [base] + quick_sort(right)
```

冒泡排序

原理：遍历需要排序的序列元素，依次比较两个相邻的元素，如果他们的顺序错误就进行交换。重复遍历直到没有相邻元素需要交换，即该序列已经排序完成。

```
def bubble_sort(data):
    """
    冒泡排序
    """
    if len(data) <= 1:
        return data
    for i in range(len(data) - 1):
        for j in range(len(data) - i - 1):
            if data[j] > data[j + 1]:
                data[j], data[j + 1] = data[j + 1], data[j]
    return data
```

选择排序

原理：一共需要遍历n-1次，没遍历一次选择出最小的元素，完成排序。

```
def select_sort(data):
    """
    选择排序
    """
    if len(data) <= 1:
        return data
    for i in range(len(data) - 1):
        min_index = i
        for j in range(i + 1, len(data)):
            if data[j] < data[min_index]:
                min_index = j
        data[i], data[min_index] = data[min_index], data[i]
    return data
```

归并排序

原理：先使每个子序列有序，再使子序列段间有序。

```python
def merge_sort(data):
    """
    归并排序
    """
    def merge(left, right):
        """
        归并过程
        """
        result = []  # 保存归并后的结果
        i = j = 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1
        result = result + left[i:] + right[j:]
        return result

    # 递归过程
    if len(data) <= 1:
        return data
    mid = len(data) // 2
    left = merge_sort(data[:mid])
    right = merge_sort(data[mid:])
    return merge(left, right)
```

插入排序

原理：通过构建有序序列，对未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入

```python
def insert_sort(data):
    """
    插入排序
    """
    if len(data) <= 1:
        return data
    for i in range(1, len(data)):
        for j in range(i, 0, -1):
            if data[j] < data[j - 1]:
                data[j], data[j - 1] = data[j - 1], data[j]
            else:
                break
    return data
```

希尔排序

原理：整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，它与插入排序的不同之处在于，它会优先比较距离较远的元素。

```
def shell_sort(data):
    """
    希尔排序
    """
    if len(data) <= 1:
        return data
    gap = 1
    while gap < len(data) // 3:
        gap = gap * 3 + 1
    while gap >= 1:
        for i in range(gap, len(data)):
            j = i
            while j >= gap and data[j] < data[j - gap]:
                data[j], data[j - gap] = data[j - gap], data[j]
                j -= gap
        gap //= 3
    return data
```

堆排序

原理：在堆的数据结构中，堆中的最大值总是位于根节点，把序列放入堆数据中一直维持最大堆积性质。

```
def heap_sort(data):
    """
    堆排序
    """
    def sift_down(start, end):
        """
        最大堆调整
        """
        root = start
        while True:
            child = 2 * root + 1
            if child > end:
                break
            if child + 1 <= end and data[child] < data[child + 1]:
                child += 1
            if data[root] < data[child]:
                data[root], data[child] = data[child], data[root]
                root = child
            else:
                break

    # 创建最大堆
    for start in range((len(data) - 2) // 2, -1, -1):
```

```
        sift_down(start, len(data) - 1)

    # 堆排序
    for end in range(len(data) - 1, 0, -1):
        data[0], data[end] = data[end], data[0]
        sift_down(0, end - 1)
    return data
```

桶排序

原理：桶排序是计数排序的升级版，它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。

```
def bucket_sort(data, bucket_size = 5):
    """
    桶排序
    默认5个桶
    """
    max_num, min_num = max(data), min(data)
    bucket_count = (max_num - min_num) // bucket_size + 1
    buckets = []
    for i in range(bucket_count):
        buckets.append([])
    for num in data:
        buckets[(num - min_num) // bucket_size].append(num)
    data.clear()
    for bucket in buckets:
        insert_sort(bucket)
        data.extend(bucket)
    return data
```

计数排序

原理：先开辟一个覆盖序列范围的数组空间，将输入的数据值转化为键存储在额外开辟的数组空间中，再依次取出。

```
def count_sort(data):
    """
    计数排序
    需要是整数序列
    """
    if len(data) <= 1:
        return data
    bucket = [0] * (max(data) + 1)
    for num in data:
        bucket[num] += 1
    i = 0
    for j in range(len(bucket)):
        while bucket[j] > 0:
```

```
            data[i] = j
            bucket[j] -= 1
            i += 1
    return data
```

基数排序

原理：将序列所有数字统一为相同数字长度，数字较短的数前面补零。从最低位开始，依次进行一次排序，然后从低位到高位依次完成排序。

```
def radix_sort(data):
    """
    基数排序
    """
    if len(data) <= 1:
        return data
    radix = 10
    div = 1
    max_bit = len(str(max(data)))
    bucket = [[] for i in range(radix)]
    while max_bit:
        for num in data:
            bucket[num // div % radix].append(num)
        j = 0
        for b in bucket:
            while b:
                data[j] = b.pop(0)
                j += 1
        div *= 10
        max_bit -= 1
    return data
```

测试结果

```
data = [2,3,5,7,1,4,6,15,5,2,7,9,10,15,9,17,12]
print("input data {}".format(data))
quick_ret = quick_sort(data)  # 快速排序
print("quick_ret {}".format(quick_ret))
bubble_ret = bubble_sort(data)  # 冒泡排序
print("bubble_ret {}".format(bubble_ret))
select_ret = select_sort(data)  # 选择排序
print("select_ret {}".format(select_ret))
merge_ret = merge_sort(data)  # 归并排序
print("merge_ret {}".format(merge_ret))
insert_ret = insert_sort(data)  # 插入排序
print("insert_ret {}".format(insert_ret))
shell_ret = shell_sort(data)  # 希尔排序
print("shell_ret {}".format(shell_ret))
```

```
heap_ret = heap_sort(data)  # 堆排序
print("heap_ret {}".format(heap_ret))
bucket_ret = bucket_sort(data)  # 桶排序
print("bucket_ret {}".format(bucket_ret))
count_ret = count_sort(data)  # 计数排序
print("count_ret {}".format(count_ret))
radix_ret = radix_sort(data)  # 基数排序
print("radix_ret {}".format(radix_ret))

input data [2, 3, 5, 7, 1, 4, 6, 15, 5, 2, 7, 9, 10, 15, 9, 17, 12]
quick_ret [1, 2, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
bubble_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
select_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
merge_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
insert_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
shell_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
heap_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
bucket_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
count_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
radix_ret [1, 2, 3, 4, 5, 5, 6, 7, 7, 9, 9, 10, 12, 15, 15, 17]
```

完~