暨南大学本科上机课实验报告

学生姓名：肖健成　　学号：2021103268

学院：网络空间安全　　专业：网络空间安全

课程名称:《算法分析与设计》　　实验名称：　　实现快排序算法

实验类型：验证-设计-综合　　实验时间：2023 年 10 月 20 日晚　　地点：517

上机内容：
（1）设计利用递归与分治策略设计简单的冒泡排序、快速排序算法;
（2）设计具有数据输入、处理和输出功能的快速排序算法，实现算法的编译和运行记录实验过程并整理实验结果。

算法描述：
冒泡排序是一种简单而基本的比较排序算法，其目标是将一个无序的数组按照升序或降序排列。该算法通过多次遍历数组，比较相邻元素的大小并交换它们，从而逐渐将较大或较小的元素“冒泡”到数组的一端。
以下是对冒泡排序算法的描述：
算法步骤：
1.从数组的第一个元素开始，逐个比较相邻的元素，将较大（或较小，取决于排序顺序）的元素向右移动。
2.重复这个过程，直到达到数组的末尾。此时，数组中最大（或最小）的元素将沉到数组的末尾位置。
3.重复以上步骤，但这次忽略已经排序好的末尾部分。继续进行相邻元素的比较和交换，将下一个最大（或最小）的元素冒泡到未排序部分的末尾。
4.重复以上操作，每次减少一个元素的未排序部分，直到整个数组排序完成。在每轮遍历中，未排序部分的长度减少一个元素，因此排序完成需要 n-1 轮遍历（n 为数组的长度）。
在实现冒泡排序的算法中，我们使用了递归的思想进行实现。

冒泡排序的伪代码
```
BubbleSort(A)
// 输入：待排序的数组 A，包含 n 个元素
// 输出：升序排列的数组 A

n = length(A)   // 获取数组长度
```

```
for i from 0 to n - 2 do:
    flag = false   // 设置标志位，表示是否发生了交换

    for j from 0 to n - 2 - i do:
        if A[j] > A[j+1] then:
            swap A[j] and A[j+1]   // 交换 A[j] 和 A[j+1] 的位置
            flag = true   // 设置标志位为 true，表示发生了交换

    if flag is false then:
        break   // 如果本轮没有发生交换，数组已经有序，提前退出

// 返回排序后的数组 A
```

快速排序（Quick Sort）是一种高效的分治排序算法，其主要思想是通过选择一个基准元素，将数组分为两个子数组，一个子数组中的元素小于基准，另一个子数组中的元素大于基准，然后递归地对这两个子数组进行排序。它被广泛应用于各种排序任务，因为它在平均情况下具有出色的性能。
以下是对快速排序算法的描述：
算法步骤：
1.选择基准元素：从数组中选择一个元素作为基准元素。通常选择中间元素、第一个元素或最后一个元素作为基准。
2.分区操作：将数组分成两个子数组，一个包含小于基准的元素，另一个包含大于基准的元素。这个过程称为分区操作。
3.递归排序：对两个子数组递归地应用快速排序算法，直到子数组变得足够小，可以被直接排序。
4.合并结果：最后，将排序好的子数组合并在一起，形成一个完全有序的数组。
在实现快速排序的算法中，我们使用了递归，分治的思想进行实现。

快速排序的伪代码

```
QuickSort(arr, low, high)
// 输入：待排序数组 arr，排序范围的起始索引 low 和结束索引 high
// 输出：升序排列的数组 arr

if low < high then:
    // 选择中间元素作为基准
    pivot = arr[low + (high - low) / 2]

    // 进行分区操作，返回分区后的基准位置
    index = Partition(arr, low, high, pivot)

    // 递归排序分区的左侧和右侧
    QuickSort(arr, low, index - 1)
    QuickSort(arr, index + 1, high)
```

```
Partition(arr, low, high, pivot)
// 分区操作：将数组 arr 分成两部分，小于等于基准的元素和大于基准的元素
// 输入：待分区的数组 arr，排序范围的起始索引 low 和结束索引 high，基准元素 pivot
// 输出：分区后的基准位置 index

i = low
j = high

while i <= j do:
    // 在左侧找到大于等于基准的元素
    while arr[i] < pivot do:
        i = i + 1

    // 在右侧找到小于等于基准的元素
    while arr[j] > pivot do:
        j = j - 1

    if i <= j then:
        // 交换 arr[i] 和 arr[j]
        Swap(arr[i], arr[j])
        i = i + 1
        j = j - 1

// 返回基准位置
return i
```

源程序：

实现冒泡排序方法 1：递归实现

```python
def recursive_bubble_sort(arr, n):
    """
    使用递归实现的冒泡排序算法，对给定列表进行升序排序。

    Args:
        arr (list): 待排序的列表
        n (int): 列表的长度

    Returns:
        None (in-place 排序)
```

```
    """

    # 基本情况：当只有一个元素或没有元素需要排序时，返回
    if n <= 1:
        return

    # 一次遍历：将最大的元素移动到列表末尾
    for i in range(n - 1):
        if arr[i] > arr[i + 1]:
            arr[i], arr[i + 1] = arr[i + 1], arr[i]

    # 递归调用：对剩余的元素进行排序
    recursive_bubble_sort(arr, n - 1)

# 示例用法
if __name__ == "__main__":
    unsorted_list = [54, 34, 25, 12, 22, 18, 90]
    print("排序前的列表：", unsorted_list)
    # 调用冒泡排序函数对列表进行排序
    recursive_bubble_sort(unsorted_list, len(unsorted_list))

    # 打印排序后的列表
    print("排序后的列表：", unsorted_list)
```

实现快速排序方法 1：递归实现

```
def quick_sort(arr):
    """
    使用快速排序算法对给定列表进行升序排序。

    Args:
        arr (list): 待排序的列表

    Returns:
        list: 排序后的列表
    """
    # 基本情况：如果列表长度小于等于 1，直接返回该列表
    if len(arr) <= 1:
        return arr

    # 选择中间元素作为基准
    pivot = arr[len(arr) // 2]

    # 利用列表推导将元素分为三部分
    left = [x for x in arr if x < pivot]  # 所有小于基准的元素
```

```
    middle = [x for x in arr if x == pivot]  # 所有等于基准的元素
    right = [x for x in arr if x > pivot]  # 所有大于基准的元素

    # 递归排序左侧和右侧的子数组
    left = quick_sort(left)
    right = quick_sort(right)

    # 将左侧、中间和右侧部分合并成一个排序好的列表
    return left + middle + right

# 示例用法
if __name__ == "__main__":
    arr = [3, 6, 8, 23, 24, 76, 10, 1, 2, 1]
    print("排序前的数组:", arr)
    sorted_arr = quick_sort(arr)
    print("排序后的数组:", sorted_arr)
```

实现快速排序方法 2：分治实现

```
def quick_sort(arr):
    """
    该函数实现快速排序算法，采用分治策略。它接受一个未排序的数组作为输入，将其排序并返回有序数组。

    Args:
        arr (list): 待排序的输入数组。

    Returns:
        list: 排序后的数组。

    """

    # 基本情况：如果数组长度小于等于 1，它已经是有序的，直接返回
    if len(arr) <= 1:
        return arr

    # 选择中间元素作为基准元素
    pivot = arr[len(arr) // 2]

    # 初始化左、中、右子数组
    left = [x for x in arr if x < pivot]  # 存储所有小于基准的元素
    middle = [x for x in arr if x == pivot]  # 存储所有等于基准的元素
    right = [x for x in arr if x > pivot]  # 存储所有大于基准的元素
```

```
    # 递归排序左子数组和右子数组
    return quick_sort(left) + middle + quick_sort(right)

# 测试快速排序
arr = [3, 6, 8, 23, 24, 76, 10, 1, 2, 1]
print("排序前的数组:", arr)
sorted_arr = quick_sort(arr)
print("排序后的数组:", sorted_arr)
```

设计具有数据输入、处理和输出功能的快速排序算法

```
import sys

def quick_sort(arr):
    """
    使用快速排序算法对给定列表进行升序排序。

    Args:
        arr (list): 待排序的列表

    Returns:
        list: 排序后的列表
    """
    # 基本情况：如果列表长度小于等于 1，直接返回该列表
    if len(arr) <= 1:
        return arr

    # 选择中间元素作为枢纽元
    pivot = arr[len(arr) // 2]

    # 利用列表推导将元素分为三部分
    left = [x for x in arr if x < pivot]  # 所有小于枢纽元的元素
    middle = [x for x in arr if x == pivot]  # 所有等于枢纽元的元素
    right = [x for x in arr if x > pivot]  # 所有大于枢纽元的元素

    # 递归排序左侧和右侧的子数组
    left = quick_sort(left)
    right = quick_sort(right)

    # 将左侧、中间和右侧部分合并成一个排序好的列表
    return left + middle + right

def main():
    try:
        if len(sys.argv) > 1:
```

```
            input_list = [int(x) for x in sys.argv[1:]]
        else:
            input_str = input("请输入用空格分隔的整数列表: ")
            input_list = [int(x) for x in input_str.split()]

        print("排序前的数组:", input_list)
        sorted_list = quick_sort(input_list)
        print("排序后的数组:", sorted_list)
    except ValueError:
        print("无效的输入，请输入整数列表，用空格分隔。")

if __name__ == "__main__":
    main()
```

复杂度分析：
①冒泡排序
时间复杂度：
最佳情况：冒泡排序在最佳情况下的时间复杂度为 O(n)，其中 n 为待排序数组的长度。这种情况发生在输入数据已经完全有序的情况下。在这种情况下，虽然需要进行多次比较，但没有发生交换操作，所以每轮排序只需要一次比较。
最坏情况：冒泡排序在最坏情况下的时间复杂度为 O(n^2)。最坏情况发生在输入数据是逆序排列的情况下，每一轮排序都需要进行 n 次比较和可能的 n 次交换。
平均情况：在平均情况下，冒泡排序的时间复杂度也是 O(n^2)。这是因为无论数据的初始顺序如何，都需要进行 n-1 轮排序操作。每一轮排序的比较和交换次数都逐渐减少，但平均下来仍然是 O(n^2)。
空间复杂度：
冒泡排序是一种原地排序算法，它只需要少量额外的内存空间用于交换元素，所以空间复杂度为 O(1)。
②快速排序算法
时间复杂度：
最佳情况：在最佳情况下，快速排序的时间复杂度为 O(n log n)，其中 n 为待排序数组的长度。最佳情况发生在每次选择的基准元素都能将数组均匀地分为两个子数组，每个子数组都包含相等数量的元素。这样的分割使得递归深度最小，排序效率最高。
最坏情况：在最坏情况下，快速排序的时间复杂度为 O(n^2)。最坏情况发生在每次选择的基准元素都是数组中的最大或最小元素，导致数组分割极不均匀。这样的分割使得递归深度最大，排序效率最低。

平均情况：在平均情况下，快速排序的时间复杂度仍然为 O(n log n)。这是因为平均情况下，基准元素的选择是随机的或者足够均匀的，导致递归深度保持在较低水平，排序效率相对较高。
空间复杂度：
快速排序是一种原地排序算法，它只需要常量级别的额外内存空间用于递归调用的栈。因此，空间复杂度为 O(log n)，其中 log n 为递归调用的深度。

程序的运行结果及截图：
实现冒泡排序方法 1：递归实现

```
排序前的列表： [54, 34, 25, 12, 22, 18, 90]
排序后的列表： [12, 18, 22, 25, 34, 54, 90]
```

实现快速排序方法 1：递归实现

```
排序前的数组: [3, 6, 8, 23, 24, 76, 10, 1, 2, 1]
排序后的数组: [1, 1, 2, 3, 6, 8, 10, 23, 24, 76]
```

实现快速排序方法 2：分治实现

```
排序前的数组: [3, 6, 8, 23, 24, 76, 10, 1, 2, 1]
排序后的数组: [1, 1, 2, 3, 6, 8, 10, 23, 24, 76]
```

设计具有数据输入、处理和输出功能的快速排序算法
正确输入

```
请输入用空格分隔的整数列表: 1 4 2 3 5 7
排序前的数组: [1, 4, 2, 3, 5, 7]
排序后的数组: [1, 2, 3, 4, 5, 7]
```

错误输入下的错误反馈机制

```
请输入用空格分隔的整数列表: 1 w 2 3
无效的输入，请输入整数列表，用空格分隔。
```

个人总结：
冒泡排序（Bubble Sort）：
冒泡排序是一种简单的排序算法，其核心思想是反复交换相邻的元素，将较大（或较小）的元素逐渐“冒泡”到数组的末尾（或开头）。
冒泡排序是一种稳定的排序算法，即相同元素的相对顺序在排序后不会改变。
时间复杂度：
1.最佳情况：O(n)，当输入数据已经有序时。
2.最坏情况：O(n^2)，当输入数据是逆序的时。
3.平均情况：O(n^2)。
空间复杂度：O(1)，冒泡排序是一种原地排序算法。
冒泡排序对于小规模数据集或接近有序的数据集是一个合适的选择，但在大规模数据集上性能较差。
快速排序（Quick Sort）：
快速排序是一种高效的分治排序算法，其核心思想是选择一个基准元素，将数组分成左右两部分，左边部分的元素小于等于基准，右边部分的元素大于基准，然后递归地对左右两部分进行排序。
快速排序是不稳定的排序算法，即相同元素的相对顺序在排序后可能会改变。
时间复杂度：
1.最佳情况：O(n log n)，当每次基准的选择都能均匀地划分数组时。
2.最坏情况：O(n^2)，当每次基准的选择都是最大或最小元素时。
3.平均情况：O(n log n)。
空间复杂度：O(log n)，快速排序的递归调用栈的深度。
快速排序通常表现出色，特别适用于大规模数据集。然而，需要谨慎处理递归深度，以避免栈溢出问题。
总结：冒泡排序是一种简单但性能较差的排序算法，适用于小规模数据集。快速排序是一种高效的排序算法，特别适用于大规模数据集。理解这两种算法的原理和性能特点有助于我们在不同情况下选择适当的排序算法。