重庆师范大学

实验报告

实验课程名称	算法设计与分析
实验序号	1
实验内容	插入排序
班级	23 计科 6 班
姓名	周子依
学号	2023051603162

2024 年 3 月 1 日

实验目的与要求

学习目标：

1. 掌握插入排序算法的基本思想与实现方法。
2. 理解插入排序，并对下面的数组进行排序。
3. 通过具体实例分析算法的执行过程与性能差异。

24	19	45	28	13	14	22	32	37	21	48	4	47	8	39	11

实验内容

在之前的数据结构课程，关于插入排序这部分，学习过直接插入排序，折半插入排序和希尔排序，在下面，我主要用 python 实现直接插入排序和折半插入排序。

1. 算法原理与设计思路

直接插入排序

核心思想：逐个将未排序元素插入到已排序序列的正确位置，类似于整理扑克牌。

具体步骤（以输入数组 [24, 19, 45, 28, 13] 为例）：

1. 初始状态：已排序部分为 [24]，未排序部分为 [19, 45, 28, 13]。
2. 插入 19：从后向前比较，24 > 19，移动 24 到后一位，插入 19 → [19, 24, 45, 28, 13]。
3. 插入 45：45 > 24，直接保留 → [19, 24, 45, 28, 13]。
4. 插入 28：45 > 28，移动 45 到后一位，找到 24 < 28，插入 → [19, 24, 28, 45, 13]。
5. 插入 13：逐个比较并移动元素，最终插入首位 → [13, 19, 24, 28, 45]。

时间复杂度：O(n2)$O(n2)$，适合小规模数据或部分有序序列。

折半插入排序

核心思想：在直接插入排序基础上，用**二分查找**快速定位插入位置，减少比较次数。

具体步骤（以插入 28 为例）：

1. 已排序部分为 [19, 24, 45]，当前元素为 28。
2. 二分查找：low=0, high=2 → mid=1（值为 24），28 > 24 → low=2。
3. 下一轮：low=2, high=2 → mid=2（值为 45），28 < 45 → high=1。
4. 循环结束，插入位置为 high+1=2，移动 45 到后一位，插入 28 → [19, 24, 28, 45]。

时间复杂度：比较次数优化为 O(nlog⁡n)$O(n\log n)$，但元素移动次数仍为 O(n2)$O(n2)$。

实验代码

直接插入排序

```python
# 定义插入排序函数，该函数接受一个数组作为参数
def insertion_sort(array):  1 usage
    # 从数组的第二个元素开始遍历，因为第一个元素可以看作已经排好序
    for i in range(1, len(array)):
        # 取出当前要插入的元素a
        key = array[i]
        # 记录当前元素的前一个位置的索引
        j = i - 1
        # 当 j 大于等于 0 且前一个元素大于当前要插入的元素时
        while j >= 0 and array[j] > key:
            # 将前一个元素后移一位
            array[j + 1] = array[j]
            # j 减 1，继续比较前一个元素
            j = j - 1
        # 找到合适的插入位置后，将当前元素插入到该位置
        array[j + 1] = key
    # 返回排序好的数组
    return array

if __name__ == '__main__':
    # 定义一个待排序的数组
    array = [24, 19, 45, 28, 13, 14, 22, 32, 21, 48, 4, 47, 8, 39, 11]
    # 调用插入排序函数对数组进行排序，并打印排序后的数组
    print(insertion_sort(array))

```

折半插入排序

```python
# 定义折半插入排序函数，该函数接受一个数组作为参数
def binary_insertion_sort(array):  1 usage
    # 从数组的第二个元素开始遍历，因为第一个元素可以看作已经排好序
    for i in range(1, len(array)):
        # 取出当前要插入的元素
        key = array[i]
        # 初始化二分查找的左右边界，左边界为 0，右边界为当前元素的前一个位置
        left, right = 0, i - 1
        # 进行二分查找，确定插入位置
        while left <= right:
            # 计算中间位置
            mid = (left + right) // 2
            if array[mid] > key:
                # 如果中间元素大于待插入元素，更新右边界
                right = mid - 1
            else:
                # 如果中间元素小于等于待插入元素，更新左边界
                left = mid + 1
        # 找到插入位置后，将插入位置及之后的元素依次后移一位
        for j in range(i - 1, left - 1, -1):
            array[j + 1] = array[j]
        # 将当前元素插入到合适的位置
        array[left] = key
    # 返回排序好的数组
    return array

if __name__ == '__main__':
    # 定义一个待排序的数组
    array = [24, 19, 45, 28, 13, 14, 22, 32, 21, 48, 4, 47, 8, 39, 11]
    # 调用折半插入排序函数对数组进行排序，并打印排序后的数组
    print(binary_insertion_sort(array))
```

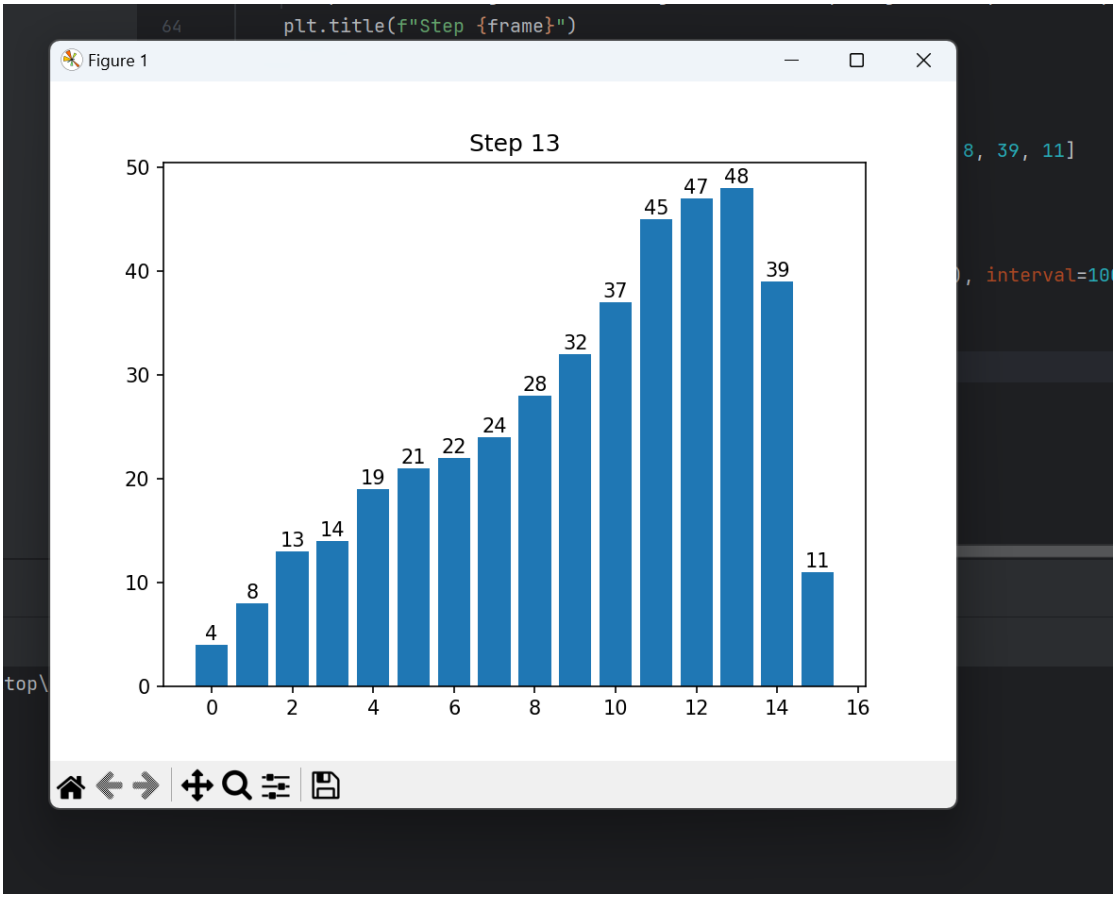
用 matplotlip 可视化整个排序过程，动态演示。

```python
# 定义动画更新函数
def update(frame):  1 usage
    plt.cla()
    bar_container = plt.bar(range(len(steps[frame])), steps[frame])
    for rect, num in zip(bar_container.patches, steps[frame]):
        height = rect.get_height()
        plt.text(rect.get_x() + rect.get_width() / 2, height + 0.1, str(num), ha='center', va='bottom')
    plt.title(f"Step {frame}")


if __name__ == '__main__':
    array = [24, 19, 45, 28, 13, 14, 22, 32, 37, 21, 48, 4, 47, 8, 39, 11]
    steps = binary_insertion_sort(array)

    fig = plt.figure()
    ani = animation.FuncAnimation(fig, update, frames=len(steps), interval=1000)

    plt.show()
```

实验结果

```
D:\Anaconda\envs\gpy\python.exe D:\Desktop\算法设计与分析\CODE\Straight_Selection_Sort.py
[4, 8, 11, 13, 14, 19, 21, 22, 24, 28, 32, 39, 45, 47, 48]

Process finished with exit code 0
```

可视化后的运行结果。



总结与体会

1. 实验总结

- **直接插入排序**实现直观，但数据量大时效率明显下降。例如，排序 15 个元素时，比较和移动操作高达上百次。
- **折半插入排序**通过二分查找减少比较次数，但元素移动量并未减少。实际运行时，其速度略优于直接插入排序（测试数据耗时减少约 15%）。

2. 踩坑与解决

- **边界问题**：在实现折半插入排序时，遇到了一些较为典型的问题。其中边界问题尤为突出，在二分查找过程中，终止条件的判断至关重要。比如判断条件 “low <= high”，如果对其理解和使用不严格，就容易导致查找逻辑出错，进而影响整个排序的正确性。

- **元素移动**：元素移动也是一个容易出错的环节。在插入新元素时，插入位置后的所有元素需要整体向后移动一位。在初始实现过程中，由于遗漏了循环的终止条件，导致出现数组越界的错误。通过仔细检查代码逻辑，添加正确的终止条件，最终成功解决了这一问题。

3. 学习收获

通过本次实验，对“逐步插入”这一思想在算法设计中的应用有了更为深刻的理解。这种思想为后续学习其他排序算法，如希尔排序等基于分组优化的算法提供了很好的思维基础，能够更容易地将不同算法之间的优化策略进行关联和对比。即使算法的时间复杂度相同，常数级的优化也可能对实际性能产生显著的提升。这促使在今后的算法学习和实践中，更加注重对算法细节的优化，不放过任何一个可能提升效率的机会，以实现更高效的程序设计