习题三

202328015926048-丁力

1. 编写程序实现归并排序算法 MergeSort 和快速排序算法 QuickSort;

好，我们这里分别实现这两者排序的算法：

MergeSort

mergeSort的原理就是通过divide and conquer的思路来解决的，通过二分的拆分数组，然后得到长度为1的数组，然后分别进行升序合并，最后就可以得到一个排序的好的数组。

分别需要实现mergeSort的递归程序和merge这个函数，首先我们借助于Python来实现（因为我比较熟悉Python)：

```
class MergeSort:
"""
Implementation of the merge sort algorithm.
"""

@staticmethod
def merge(array: list[int], start: int, mid: int, end: int) -> list[int]:
  """
  Merge two sorted sub-arrays into one.

  Parameters:
        - array: The array containing the two sub-arrays.
        - start: The starting index of the first sub-array.
        - mid: The ending index of the first sub-array.
        - end: The ending index of the second sub-array.

        Returns:
        - The merged array.
        """
        left_length = mid - start + 1
        right_length = end - mid
        left_subarray = [None] * (left_length + 1)
        right_subarray = [None] * (right_length + 1)

        for i in range(left_length):
            left_subarray[i] = array[start + i]
        for i in range(right_length):
            right_subarray[i] = array[mid + i + 1]

        # Sentinel values
        left_subarray[left_length] = float('inf')
        right_subarray[right_length] = float('inf')

        i, j = 0, 0
        for k in range(start, end + 1):
```

```
            if left_subarray[i] <= right_subarray[j]:
                array[k] = left_subarray[i]
                i += 1
            else:
                array[k] = right_subarray[j]
                j += 1

        return array

    def sort(self, array: list[int], start_pos: int = 0, end_pos: int = None) -> list[int]:
        """
        Sort the array using the merge sort algorithm.

        Parameters:
        - array: The array to be sorted.
        - start_pos: The starting index of the array segment to be sorted (default is 0).
        - end_pos: The ending index of the array segment to be sorted (default is the last
element).

        Returns:
        - The sorted array.
        """
        if end_pos is None:
            end_pos = len(array) - 1

        if start_pos < end_pos:
            mid_pos = (start_pos + end_pos) // 2  # Integer division
            self.sort(array, start_pos, mid_pos)
            self.sort(array, mid_pos + 1, end_pos)
            self.merge(array, start_pos, mid_pos, end_pos)

        return array
```

QuickSort

同样的，这里我们也能实现`quickSort`：

```
class QuickSort:
    """
    Implementation of the quicksort algorithm.
    """

    @staticmethod
    def partition(array: list[int], start: int, end: int) -> int:
        """
        Partition the array segment into two sub-segments.

        Parameters:001.
        - array: The array segment to be partitioned.
        - start: The starting index of the array segment.
        - end: The ending index of the array segment.

        Returns:
        - The index of the pivot after partitioning.
        """
        pivot = array[end]
        i = start - 1
        for j in range(start, end):
            if array[j] <= pivot:
                i += 1
                array[i], array[j] = array[j], array[i]
        array[i + 1], array[end] = array[end], array[i + 1]
        return i + 1
```

```python
    def sort(self, array: list[int], start_pos: int = 0, end_pos: int = None) -> list[int]:
        """
        Sort the array using the quicksort algorithm.

        Parameters:
        - array: The array to be sorted.
        - start_pos: The starting index of the array segment to be sorted (default is 0).
        - end_pos: The ending index of the array segment to be sorted (default is the last
element).

        Returns:
        - The sorted array.
        """
        if end_pos is None:
            end_pos = len(array) - 1

        if start_pos < end_pos:
            pivot_pos = self.partition(array, start_pos, end_pos)
            self.sort(array, start_pos, pivot_pos - 1)
            self.sort(array, pivot_pos + 1, end_pos)

        return array


if __name__ == "__main__":
    test_array = [2, 1, 45, 21, 5]
    sorter = QuickSort()
    sorted_result = sorter.sort(test_array)
    print(sorted_result)
```

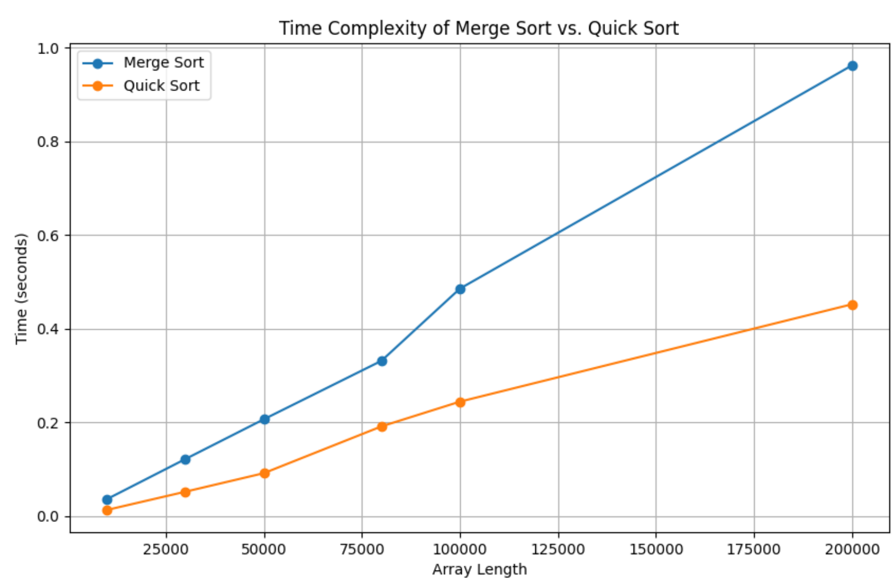
2,用长分别为 10000、30000、50000、80000、100000、200000 的 6 个数组(可用机器 随机产生)的排列来统计这两种算法的时间复杂性;

编写如下代码进行速度测试:

```python
import random
import time
import matplotlib.pyplot as plt
from merge_sort import MergeSort
from quick_sort import QuickSort

# Generate test arrays
lengths = [10000, 30000, 50000, 80000, 100000, 200000]
test_arrays = [random.sample(range(1, length * 10), length) for length in lengths]

# Time the algorithms
merge_sort_times = []
quick_sort_times = []

merge_sorter = MergeSort()
quick_sorter = QuickSort()

for test_array in test_arrays:
```

```
    # Time Merge Sort
    array_copy = test_array.copy()
    start_time = time.time()
    merge_sorter.sort(array_copy)
    merge_sort_times.append(time.time() - start_time)

    # Time QuickSort
    array_copy = test_array.copy()
    start_time = time.time()
    quick_sorter.sort(array_copy)
    quick_sort_times.append(time.time() - start_time)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(lengths, merge_sort_times, '-o', label='Merge Sort')
plt.plot(lengths, quick_sort_times, '-o', label='Quick Sort')
plt.xlabel('Array Length')
plt.ylabel('Time (seconds)')
plt.title('Time Complexity of Merge Sort vs. Quick Sort')
plt.legend()
plt.grid(True)
plt.show()
```

得到如下结果:



从上可以看出，对于我们上述的实现，随着数组长度的线性增加，排序速度基本呈现一个线性的趋势， 但是实际上两者的摊销复杂度都是$\Theta(n \ln n)$。

3.讨论归并排序算法 MergeSort 的空间复杂性。

在归并排序中，分解过程本身并不需要额外的空间（只需要递归调用栈的空间），但合并过程需要额外的空间来存储两个子数组的元素。对于长度为 n 的数组，这两个子数组的最大长度分别为 $\frac{n}{2} 和\frac{n}{2}。因此，每次合并的时候，最多需要 n $ 个额外的空间。

但是，归并排序的一个重要特性是它在任何时候都只进行一次合并操作。因此，尽管每一层的递归都需要额外的空间，但这些空间是重复使用的，而不是累加的。这意味着总的额外空间需求与原始数组的大小成正比。

结论:
归并排序的空间复杂性为 $O(n)$，其中 n 是待排序数组的长度。这是因为我们需要额外的空间来存储子数组并进行合并操作，而这些额外的空间与原数组的大小成正比。

4.

说明算法 PartSelect 的平均时间复杂性为 $O(n)$ 。
提示：假定数组中的元素各不相同，且第一次划分时划分元素 v 是第 i 小元素的概率为 $1/n$ 。因为 Partition 中的 case 语句所要求的时间都是 $O(n)$，所以，存在常数 c，使得算法 PartSelect 的平均时间复杂度 $C_A^k(n)$ 可以表示为

$$C_A^k(n) \leq cn + \frac{1}{n}\left(\sum_{1\leq i<k} C_A^{k-i}(n-i) + \sum_{k<i\leq n} C_A^k(i-1)\right)$$

令 $R(n)=\max_k\left(C_A^k(n)\right)$，取 $c \geq R(1)$，试证明 $R(n) \leq 4cn$ 。

> 这一题，我将采用数学归纳法来证明， 题干已经对 `PartSelect` 的平均时间复杂度进行了数学的等效，所以我只需要进行证明式即可。
>
> 由于$R(n)=\max_k\left(C_A^k(n)\right)$，所以：
>
> $$R(n) \leq cn + \frac{1}{n}\left(\sum_{i=1}^{n-1} R(n-i) + \sum_{i=1}^{n-1} R(i)\right)$$
>
> - 当$n=1$时，原式变为：
>
> $$R(1) \leq 4c$$
>
>   此时显然成立。
>
> - 归纳假设
>   - 假设对所有小于 n 的自然数 m ，都有 $R(m) \leq 4\,\mathrm{cm}$ 成立。
>
>   归纳步骤：
>
>   - 我们想要证明 $R(n) \leq 4cn$。根据给定的公式，我们有：
>
> $$\begin{aligned} R(n) &\leq cn + \frac{1}{n}(R(n-1)+\cdots+R\left(n-k_n+1\right)) + \left(R\left(k_n\right)+R\left(k_n+1\right)+\cdots+R(n-1)\right) \\ &\leq cn + \frac{4c}{n}((n-1)+\cdots+\left(n-k_n+1\right)) + \left(k_n+\cdots+(n-1)\right) \end{aligned}$$
>
>   - 在上述表达式中，我们利用了归纳假设 $R(m) \leq 4\,\mathrm{cm}$。
>   - 接下来，我们继续简化上述表达式：
>
> $$\begin{aligned} R(n) &\leq cn + \frac{4c}{n}\left(\frac{\left(k_n-1\right)\left(2n-k_n\right)}{2} + \frac{\left(n-k_n\right)\left(k_n+n-1\right)}{2}\right) \\ &\leq cn - \frac{4c}{n}\left(k_n^2-(n+1)k_n-\frac{n^2-3n}{2}\right) \\ &\leq cn - \frac{4c}{n}\left(-\left(\frac{n+1}{2}\right)^2-\frac{n^2-3n}{2}\right) \\ &\leq cn + c\frac{3n^2-4n+1}{n} \\ &\leq cn + c(3n-3) \\ &\leq 4cn \end{aligned}$$
>
>   - 通过上述步骤，我们得到 $R(n) \leq 4\mathrm{cn}$ ，这完成了归纳步骤。