目录

1.实现快速排序代码

快速排序（Quick Sort）是一种高效的排序算法，由C. A. R. Hoare在1960年提出。它是一种分治法（Divide and Conquer）策略的典型应用。

快速排序的原理：

1. **选择基准值（Pivot）**： 快速排序首先从数组中选择一个元素作为基准值，这个值称为“pivot”。选择的方法可以多样，如选择第一个元素、最后一个元素、中间元素或随机元素。
2. **分区操作**： 数组被分为两个部分，使得：
   - 左边部分的所有元素都不大于基准值，
   - 右边部分的所有元素都不小于基准值。

   此时，基准值处于整个数组中的最终位置。
3. **递归排序**： 递归地对基准左侧和右侧的两个子数组进行快速排序，直到子数组的长度为1或0，此时数组已经完全排序。

快速排序主要有两种实现方式，分别是递归方式和迭代方式。

下面我们首先来看一下递归方式实现的快速排序的代码：

Python代码实现（递归版本）：

以下是快速排序的一个简单Python实现，其中使用了Lomuto分区方案：

```
def quick_sort(arr):
    def partition(low, high):
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1

    def quick_sort_recursive(low, high):
        if low < high:
            pi = partition(low, high)
            quick_sort_recursive(low, pi - 1)
            quick_sort_recursive(pi + 1, high)
```

```
    quick_sort_recursive(0, len(arr) - 1)
    return arr

# 测试用例
test_cases = [
    [10, 7, 8, 9, 1, 5],
    [1, 2, 3, 4, 5],
    [5, 4, 3, 2, 1],
    [],
    [1],
    [11, 11, 11, 11]
]

# 展示排序结果
for case in test_cases:
    print(f"Original: {case}")
    print(f"Sorted: {quick_sort(case)}\n")

# 代码运行结果
Original: [10, 7, 8, 9, 1, 5]
Sorted: [1, 5, 7, 8, 9, 10]

Original: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]

Original: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]

Original: []
Sorted: []

Original: [1]
Sorted: [1]

Original: [11, 11, 11, 11]
Sorted: [11, 11, 11, 11]
```

测试用例及其输出：

上面的代码对包含多种情况的测试用例进行了排序，包括：

- 普通未排序数组，
- 已排序数组，
- 逆序数组，
- 空数组，
- 单元素数组，
- 所有元素相同的数组。

这些测试用例涵盖了快速排序可能面临的一些典型情况，并显示了算法处理这些情况的能力。每个测试用例的输出将展示原始数组和排序后的数组，以验证排序过程的正确性。

非递归（迭代）版本的快速排序可以使用一个显式的栈来模拟递归过程。这种方法避免了递归可能带来的栈溢出问题，并直观地展示了算法的控制流程。下面是如何使用栈实现快速排序的迭代版本：

Python代码实现（迭代版本）：

```python
def quick_sort_iterative(arr):
    if arr == []:
        return arr
    # 创建一个栈
    stack = []
    # 初始范围从0到数组长度减一
    stack.append(0)
    stack.append(len(arr) - 1)

    # 只要栈非空，就继续运行
    while stack:
        # 弹出 high 和 low
        high = stack.pop()
        low = stack.pop()

        # 使用Lomuto分区方案
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        # 交换pivot到正确位置
        i += 1
        arr[i], arr[high] = arr[high], arr[i]
        pi = i

        # 如果有左子数组，将其范围入栈
        if pi - 1 > low:
            stack.append(low)
            stack.append(pi - 1)

        # 如果有右子数组，将其范围入栈
        if pi + 1 < high:
            stack.append(pi + 1)
            stack.append(high)

    return arr

# 测试用例
test_cases = [
    [10, 7, 8, 9, 1, 5],
    [1, 2, 3, 4, 5],
    [5, 4, 3, 2, 1],
    [],
    [1],
    [11, 11, 11, 11]
```

```
]

# 展示排序结果
for case in test_cases:
    print(f"Original: {case}")
    print(f"Sorted: {quick_sort_iterative(case)}\n")

# 代码运行结果
Original: [10, 7, 8, 9, 1, 5]
Sorted: [1, 5, 7, 8, 9, 10]

Original: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]

Original: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]

Original: []
Sorted: []

Original: [1]
Sorted: [1]

Original: [11, 11, 11, 11]
Sorted: [11, 11, 11, 11]
```

在迭代版本的快速排序中，我们使用了栈来保存将要处理的子数组的索引。这种方法模拟了递归调用栈的行为：

- 首先，将整个数组的起始和结束索引推入栈中。
- 然后，使用一个循环，直到栈为空，在每次迭代中：
  - 从栈中弹出一个子数组的界限（`high` 和 `low`）。
  - 执行分区操作，确定 `pivot` 的最终位置。
  - 根据 `pivot` 的位置，决定是否将左子数组或右子数组的索引范围推回栈中。

这种迭代方法避免了递归的深度调用，特别是对于那些可能导致递归深度很深的大数组来说，是一个更稳定的选择。

迭代版本的快速排序在时间复杂度和空间复杂度上的表现与递归版本相似，但有一些关键的实现细节差异：

时间复杂度

- **最佳和平均情况**：对于平均分布的数据，快速排序的时间复杂度通常是 $O(n \log n)$。这是因为每次分区大约将数组分成两半，需要递归或迭代地应用这一过程大约 $\log n$ 次。
- **最坏情况**：在最坏的情况下，如果每次选择的基准都是最小或最大的元素，快速排序的时间复杂度会退化到 $O(n^2)$。这种情况在数组已经基本有序的情况下可能发生（完全正序或完全逆序），每次分区操作只能减少一个元素。

空间复杂度

- **递归版本**：递归版本的快速排序在最坏情况下的空间复杂度可以达到 $O(n)$，这是由递归调用栈深度决定的。在平均情况下，由于递归的深度接近 $\log n$，其空间复杂度是 $O(\log n)$。
- **迭代版本**：迭代版本使用一个显式的栈来存储未处理的子数组的界限。虽然这避免了函数调用的开销，但栈的空间使用仍然可以在最坏情况下达到 $O(n)$，特别是当数组几乎有序时，可能需要将许多小的子数组范围推入栈。在平均情况下，空间复杂度通常也是 $O(\log n)$，因为每次都将数组大致分成两部分。

稳定性

- **不稳定排序**：相等的元素可能由于分区而交换其原始顺序。

实用性和选择

尽管迭代版本避免了递归的潜在栈溢出问题，它在空间和时间上的复杂度与递归版本相似。选择递归还是迭代版本通常取决于具体的应用场景以及对栈溢出的考虑。迭代版本更适合于那些对栈空间使用有严格限制的环境，例如嵌入式系统或者非常大的数据集处理。

在实际应用中，可以通过随机选择基准值或使用“三数取中”法来选择基准值，以避免最坏情况的发生，从而使得快速排序的性能更加稳定。此外，对于小数组，可以切换到插入排序以提高效率，因为小数组上的插入排序可能比快速排序更快。这种组合策略在实际库中如C++的STL中被广泛应用。

2.实现Self_Attention（百度实习一面）

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SelfAttention(nn.Module):
def __init__(self, embed_size, heads):
    super(SelfAttention, self).__init__()
    self.embed_size = embed_size
    self.heads = heads
    self.head_dim = embed_size // heads

    assert self.head_dim * heads == embed_size, "Embed size needs to be divisible
by heads"

    self.values = nn.Linear(self.head_dim, self.head_dim, bias=False)
    self.keys = nn.Linear(self.head_dim, self.head_dim, bias=False)
    self.queries = nn.Linear(self.head_dim, self.head_dim, bias=False)
    self.fc_out = nn.Linear(heads * self.head_dim, embed_size)

def forward(self, values, keys, query, mask):
    N = query.shape[0]
    value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

    # Split the embedding into self.heads different pieces
    values = values.reshape(N, value_len, self.heads, self.head_dim)
    keys = keys.reshape(N, key_len, self.heads, self.head_dim)
    queries = query.reshape(N, query_len, self.heads, self.head_dim)
```

```
        values = self.values(values)
        keys = self.keys(keys)
        queries = self.queries(queries)

        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])
        if mask is not None:
            energy = energy.masked_fill(mask == 0, float("-1e20"))

        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)
        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(N,
    query_len, self.heads * self.head_dim)
        out = self.fc_out(out)
        return out

    # Example usage:
    embed_size = 256
    heads = 8
    attention_layer = SelfAttention(embed_size, heads)

    # Dummy data
    N, value_len, key_len, query_len = 3, 50, 40, 30
    value = torch.rand((N, value_len, embed_size))
    key = torch.rand((N, key_len, embed_size))
    query = torch.rand((N, query_len, embed_size))
    mask = None  # Optional mask for padded tokens

    # Forward pass
    out = attention_layer(value, key, query, mask)
    print(out.shape)  # Should be (N, query_len, embed_size)
```