Sorting Algorithm Assignment Editorial

Question : Sort Out!

```
def sortOut(arr):
    n = len(arr)
    arr1 = list(range(n))
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                # Swap the elements in arr
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                # Swap the corresponding indices in arr1
                arr1[j], arr1[j + 1] = arr1[j + 1], arr1[j]
    return arr1

def solve():
    n = int(input())
    arr = list(map(int, input().split()))
    ans = sortOut(arr)
    print(*ans)

t = 1
while t > 0:
    solve()
    t -= 1
```

Explanation

1. **Helper Function:** `sortOut(arr)`
    - **Parameters**:
        - `arr` : A list of integers whose sorted order we want to track.
    - **Process**:
        a. **Initialize**:

        ```
        n = len(arr)
        arr1 = list(range(n))
        ```

        - `n` is the length of `arr` .

- `arr1` is a list of indices from `0` to `n-1`, representing the **original positions** of the elements in `arr`.

b. **Bubble Sort**:

```
for i in range(n - 1):
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            arr1[j], arr1[j + 1] = arr1[j + 1], arr1[j]
```

- We perform a standard **bubble sort** over `arr`.
- Whenever we swap two elements in `arr`, we also swap their corresponding indices in `arr1`.
- This ensures that by the end of the sorting process, `arr1` will reflect the **original indices** of the elements in the **newly sorted** array `arr`.

c. **Return the Index List**:

```
return arr1
```

- After the array is sorted, `arr1` holds the **original** positions of the now-sorted elements.
- For example, if the smallest element was originally at index 4, but now is at the front of `arr`, then `arr1[0]` would be `4`.

2. **Main Function:** `solve()`

```
def solve():
    n = int(input())
    arr = list(map(int, input().split()))
    ans = sortOut(arr)
    print(*ans)
```

- **Read Input**:
  - `n` is the number of elements.
  - `arr` is the list of integers.
- **Sort and Retrieve Indices**:
  - We call `sortOut(arr)`, which returns a list of the **original indices** in the order that the array's elements end up after sorting.
- **Print Result**:
  - We use `print(*ans)` to print the resulting list `ans` (the indices) in a single line separated by spaces.

3. **Loop to Handle Multiple Test Cases**

```
t = 1
while t > 0:
    solve()
    t -= 1
```

- Here, `t` is set to 1, so it effectively calls `solve()` once.
- If you had multiple test cases, you could adjust `t` accordingly.

How It Works Overall

- **Goal**: Sort the array in non-decreasing order, but instead of printing the sorted elements, print their **original indices** in the new sorted order.
- **Approach**:
  i. Use **bubble sort** on `arr`.
  ii. Maintain a parallel list of indices (`arr1`).
  iii. Whenever two elements in `arr` swap, swap their corresponding indices in `arr1`.
  iv. After the sort completes, `arr1` reflects the original positions of the sorted elements.
- **Result**: When we print `arr1`, we see the original indices in the order that the elements appear in the sorted array.

Question : You have to sort it out!

```
n = int(input())
arr = list(map(int, input().split()))

indices = list(range(n))

for i in range(n - 1):
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            indices[j], indices[j + 1] = indices[j + 1], indices[j]

print(" ".join(map(str, indices)))
```

Explanation

1. **Reading the Input**

```
n = int(input())
arr = list(map(int, input().split()))
```

   - `n` is the number of elements in the array.
   - `arr` is a list of `n` integers read from a single line of input.

2. **Creating the `indices` List**

```
indices = list(range(n))
```

   - We create a list called `indices` that initially stores the integers from `0` to `n-1`.
   - The goal is to keep track of the **original positions** of the elements in `arr`.

3. **Bubble Sort**

```
for i in range(n - 1):
    for j in range(n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
            indices[j], indices[j + 1] = indices[j + 1], indices[j]
```

   - We use a **bubble sort** approach to sort the array in non-decreasing order.
   - The outer loop (`for i in range(n - 1)`) runs `n-1` times.
   - The inner loop (`for j in range(n - i - 1)`) compares adjacent elements:
     - If `arr[j]` is greater than `arr[j + 1]`, we swap them.
     - **Important**: We also swap the corresponding elements in `indices` to keep track of each element's original position.

4. **Printing the Result**

```
print(" ".join(map(str, indices)))
```

   - After sorting is complete, `indices` reflects the **original indices** of the elements in their new sorted order.

- We convert each integer in `indices` to a string and join them with spaces to produce the final output line.

How It Works Overall

- **Goal**: Sort `arr` while also outputting the **original positions** of its elements.
- **Method**: Bubble sort `arr` in place. Whenever we swap two elements in `arr`, we perform the **same swap** in `indices`.
- **Result**: Once the array is fully sorted, `indices` shows where each sorted element came from in the original array. Printing `indices` thus reveals the original indices in the sorted order of `arr`.

Question : Make Leaderboard

```
def bubbleSortOnName(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if names[j] > names[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]

def bubbleSortOnScore(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if scores[j] < scores[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]

def printLeaderBoard(names, scores):
    N = len(names)
    rank = 1
    for i in range(N):
        print(rank, names[i])
        if i != N-1 and scores[i] > scores[i+1]:
            rank = i + 2

def solve(names, scores):
    bubbleSortOnName(names, scores)
    bubbleSortOnScore(names, scores)
    printLeaderBoard(names, scores)
```

```
def inp():
    N = int(input())
    names = []
    scores = []
    for i in range(N):
        name, score = input().split()
        names.append(name)
        scores.append(int(score))
    solve(names, scores)

inp()
```

Explanation of Each Part

1. `bubbleSortOnName(names, scores)`

```
def bubbleSortOnName(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if names[j] > names[j+1]:
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]
```

- This function sorts the `names` list in **ascending** alphabetical order using a **bubble sort** approach.
- We perform `(N-1)` passes, and in each pass, we compare adjacent elements `names[j]` and `names[j+1]`.
- If `names[j]` is lexicographically **greater** than `names[j+1]`, we swap both `names` and their corresponding `scores`.
- By the end, `names` will be sorted alphabetically, and `scores` will be reordered accordingly so that each student's score remains aligned with their name.

2. `bubbleSortOnScore(names, scores)`

```
def bubbleSortOnScore(names, scores):
    N = len(names)
    for i in range(N-1):
        for j in range(N-i-1):
            if scores[j] < scores[j+1]:
```

```
                names[j], names[j+1] = names[j+1], names[j]
                scores[j], scores[j+1] = scores[j+1], scores[j]
```

- This function sorts the `scores` in **descending** order using another bubble sort pass.
- We again perform `(N-1)` passes, but this time we check if `scores[j] < scores[j+1]`.
- If so, we swap both the scores and the corresponding names to keep them aligned.
- After completion, the list will be ordered such that the highest score is at the front, moving down to the lowest score.

3. `printLeaderBoard(names, scores)`

```
def printLeaderBoard(names, scores):
    N = len(names)
    rank = 1
    for i in range(N):
        print(rank, names[i])
        if i != N-1 and scores[i] > scores[i+1]:
            rank = i + 2
```

- After sorting, this function prints out each student's **rank** and **name**.
- We start with `rank = 1`.
- For each student in the list:
  - We print the current `rank` and the student's `name`.
  - Then we check if the current student's score is greater than the **next** student's score:
    - If `scores[i] > scores[i+1]`, it means the next student has a strictly lower score, so the next student should get a new rank (i.e., `i+2`).
    - Otherwise, if the scores are the same, the next student shares the same rank.
- This logic effectively handles the case where multiple students have the same score (they share the same rank, and the next student with a lower score gets the next rank).

4. `solve(names, scores)`

```
def solve(names, scores):
    bubbleSortOnName(names, scores)
    bubbleSortOnScore(names, scores)
    printLeaderBoard(names, scores)
```

- The `solve` function orchestrates the process:
  i. Sort the list by name (alphabetically).

ii. Sort by score (descending).

iii. Print the leaderboard with the correct rank ordering.

5. `inp()` and Main Execution

```
def inp():
    N = int(input())
    names = []
    scores = []
    for i in range(N):
        name, score = input().split()
        names.append(name)
        scores.append(int(score))
    solve(names, scores)

inp()
```

- The `inp()` function:
  - Reads an integer `N` from input (number of students).
  - Initializes empty lists `names` and `scores`.
  - Repeats `N` times:
    - Reads a line, splits it into `name` (string) and `score` (integer).
    - Appends `name` to `names` and `score` to `scores`.
  - Calls the `solve(names, scores)` function to perform the sorting and ranking.
- Finally, we call `inp()` to run the entire process.

How It All Fits Together

1. The user inputs the number of students `N`, followed by `N` lines of name-score pairs.
2. `bubbleSortOnName` arranges students alphabetically, keeping each name-score pair intact.
3. `bubbleSortOnScore` then sorts these pairs by scores in descending order.
4. `printLeaderBoard` prints the rank and name for each student, adjusting the rank properly when scores differ.
5. As a result, you get a final sorted leaderboard with the highest scoring student(s) at the top, ties handled gracefully, and ranks assigned appropriately.

Question : Selection sort problem

```
# Read the number of elements N from the first line of input
N = int(input())

# Read N numbers from the second line of input and convert them to a list of integers
numbers = list(map(int, input().split()))

# Implement Selection Sort to sort the list in increasing order
for i in range(N - 1):
    # Initialize the index of the minimum element as the current position
    min_index = i

    # Find the minimum element in the unsorted portion (from i+1 to N-1)
    for j in range(i + 1, N):
        if numbers[j] < numbers[min_index]:
            min_index = j

    # Swap the current element with the minimum element found
    numbers[i], numbers[min_index] = numbers[min_index], numbers[i]

# Print the sorted list with numbers separated by spaces
print(*numbers)
```

Explanation of Each Part

1. **Reading Input**

```
N = int(input())
numbers = list(map(int, input().split()))
```

   - The first line of input gives the number of elements, `N`.
   - The second line contains `N` integers separated by spaces, which are read into the list `numbers`.

2. **Selection Sort Algorithm**

```
for i in range(N - 1):
    min_index = i
    for j in range(i + 1, N):
        if numbers[j] < numbers[min_index]:
            min_index = j
    numbers[i], numbers[min_index] = numbers[min_index], numbers[i]
```

   - **Outer loop (`i` from `0` to `N-2`)**:

        - We treat the subarray to the left of `i` as already sorted, and we will place the correct element for position `i` during each iteration.
    - `min_index = i`:
        - We assume that the element at index `i` is the minimum among the unsorted portion.
    - **Inner loop (`j` from `i+1` to `N-1`)**:
        - We scan the rest of the list (the unsorted part) to find the actual minimum element.
        - If we find an element smaller than `numbers[min_index]`, we update `min_index` to that index.
    - **Swapping**:
        - After the inner loop, the position `min_index` holds the index of the smallest element in the unsorted portion.
        - We swap that element with the element at index `i`.

3. **Printing the Sorted List**

```
print(*numbers)
```

    - The final list `numbers` is now sorted in non-decreasing order.
    - We use `print(*numbers)` to unpack the list elements and print them separated by spaces.

How Selection Sort Works

- In each iteration `i`, the algorithm selects the **minimum** element from the subarray `numbers[i ... N-1]` and swaps it into position `i`.
- By the end of the first iteration, the smallest element is in the correct place at `numbers[0]`.
- By the end of the second iteration, the second smallest is in the correct place at `numbers[1]`, and so on.
- After `N-1` such iterations, the entire array is sorted in ascending order.

Time Complexity: $O(N^2)$
Space Complexity: O(1) (sorting in place)

Question : Bubble Sort Problem

```
n = int(input())
l = list(map(int, input().split()))
```

```
for i in range(n):
    for j in range(0, n - i - 1):
        if l[j] > l[j + 1]:
            l[j], l[j + 1] = l[j + 1], l[j]

print(*l)
```

Explanation

1. **Reading Input**

```
n = int(input())
l = list(map(int, input().split()))
```

   - `n` is the number of elements in the array.
   - `l` is the list of unsorted numbers, obtained by splitting the input string into substrings and mapping each to an integer.

2. **Outer Loop**

```
for i in range(n):
```

   - This loop runs `n` times. Each pass places the next-largest element in its correct position at the end of the list.

3. **Inner Loop**

```
for j in range(0, n - i - 1):
    if l[j] > l[j + 1]:
        l[j], l[j + 1] = l[j + 1], l[j]
```

   - In each pass of the outer loop, the inner loop compares adjacent elements (`l[j]` and `l[j + 1]`) and swaps them if they are in the wrong order (`l[j] > l[j + 1]`).
   - The expression `range(0, n - i - 1)` ensures that with each outer loop pass, we do not re-check the already sorted elements at the end of the list.
   - By the end of the first pass, the largest element “bubbles up” to the last position. By the end of the second pass, the second-largest is in its place, and so on.

4. **Printing the Result**

```
print(*l)
```

- The asterisk (`*`) unpacks the list, printing its elements separated by spaces in ascending order.

Bubble Sort works by repeatedly swapping adjacent elements if they are out of order, ensuring that with each pass, the largest remaining element is moved to its correct position at the end of the list.

Question : New Sorting Algorithm

```
def solve(N, arr, K):
    arr.sort(key=lambda x: x % K)
    print(" ".join(map(str, arr)))
```

Explanation

1. **Function Signature**

```
def solve(N, arr, K):
```

- The function takes three parameters:
  - `N` : the number of elements in the array.
  - `arr` : the array (list) of integers to be sorted.
  - `K` : the integer used to define the sorting condition.

2. **Sorting the Array**

```
arr.sort(key=lambda x: x % K)
```

- We use Python's built-in `sort()` method on `arr` .
- The `key` parameter is set to a **lambda** function: `lambda x: x % K` .
- This means that each element `x` in `arr` is compared based on its remainder when divided by `K` .
- Elements with smaller remainders (`x % K`) come before those with larger remainders.

3. **Printing the Result**

```
print(" ".join(map(str, arr)))
```

- After sorting, we convert each element in `arr` to a string.
- We then join these string representations with a space separator, producing a single line of output.

How It Works Overall

- The code sorts the array `arr` **not** by the usual numerical order, but rather by each element's **remainder** when divided by `K`.
- For example, if `K = 5`, then the element `12` (which has a remainder of `2` when divided by `5`) will appear before `23` (remainder `3`), and so on.
- Finally, the sorted array is printed in one line, separated by spaces.

Question : Three Max, Three Min Please

```
def solve(N, arr):
    if N == 0:
        print("Not Possible")
        print("Not Possible")
        return

    # Remove duplicates and sort the array
    unique_values = sorted(set(arr))

    # Check for the 3 smallest values
    if len(unique_values) >= 3:
        min_values = unique_values[:3]
        print(" ".join(map(str, min_values)))
    else:
        print("Not Possible")

    # Check for the 3 largest values
    if len(unique_values) >= 3:
        max_values = unique_values[-3:]
        print(" ".join(map(str, max_values)))
    else:
        print("Not Possible")
```

Explanation

1. **Function Signature**

```
def solve(N, arr):
```

   - The function takes:
     - `N` : the number of elements in the array.
     - `arr` : the list of integers itself.

2. **Immediate Check for Empty Array**

```
if N == 0:
    print("Not Possible")
    print("Not Possible")
    return
```

   - If the array is empty (`N == 0`), there are no elements to analyze.
   - We print `"Not Possible"` twice (once for the 3 smallest values and once for the 3 largest) and then return immediately.

3. **Remove Duplicates and Sort**

```
unique_values = sorted(set(arr))
```

   - We convert `arr` into a set to remove any **duplicate** values.
   - Then we convert it back to a list and sort it in ascending order, storing the result in `unique_values` .

4. **Checking the 3 Smallest Values**

```
if len(unique_values) >= 3:
    min_values = unique_values[:3]
    print(" ".join(map(str, min_values)))
else:
    print("Not Possible")
```

   - If `unique_values` has **at least 3 distinct** elements, the first three (i.e., `unique_values[:3]`) are the 3 smallest distinct values.
   - We print them separated by spaces.

- If there are fewer than 3 distinct values, we print `"Not Possible"`.

5. **Checking the 3 Largest Values**

```
if len(unique_values) >= 3:
    max_values = unique_values[-3:]
    print(" ".join(map(str, max_values)))
else:
    print("Not Possible")
```

   - Similarly, if there are at least 3 distinct values, the last three elements of `unique_values` (`unique_values[-3:]`) are the 3 largest distinct values.
   - We print those values, also separated by spaces.
   - Otherwise, we print `"Not Possible"` again.

How It Works Overall

- The code first checks for an empty array. If it's empty, it's immediately "Not Possible" to find three minimum or maximum values.
- Otherwise, it ensures we only consider **distinct** values (by using a set) and sorts them in ascending order.
- The first three elements in this sorted list (if they exist) are the 3 smallest distinct values, and the last three elements (if they exist) are the 3 largest distinct values.
- If at any point we have fewer than 3 distinct elements, we print `"Not Possible"` for that case.