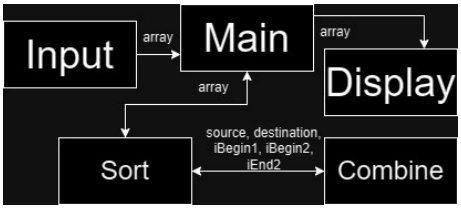
Modularization Metrics



Response

main:

Cohesion: Moderate Cohesion

Coupling: Low Coupling

Justification: main appears to serve as the central control function. It communicates with other functions but doesn't perform a specific, focused task. It's responsible for coordinating the flow of data between the other functions, which gives it moderate cohesion. However, it has low coupling because it interacts with other functions in a controlled and limited way.

input:

Cohesion: High Cohesion

Coupling: Low Coupling

Justification: input has high cohesion because its sole purpose is to read and provide input data to the main function. It doesn't perform any sorting or complex operations, focusing solely on its input task. It has low coupling because it interacts with main in a straightforward and limited manner.

sort:

Cohesion: Low Cohesion

Coupling: Moderate Coupling

Justification: sort performs the sorting operation, which is its primary responsibility, but it also interacts with the combine function. This dual functionality reduces its cohesion. It has moderate coupling because it relies on combine to perform part of the sorting process.

combine:

Cohesion: Moderate Cohesion

Coupling: Low Coupling

Justification: combine is responsible for combining two subarrays during the sorting process. It has a single, well-defined purpose, which provides it with moderate cohesion. It has low coupling because it interacts with sort in a controlled and focused manner, facilitating the sorting process.

display:

Cohesion: High Cohesion

Coupling: Low Coupling

Justification: display is focused solely on displaying the contents of an array. It has high cohesion because its purpose is well-defined. It has low coupling because it communicates with main in a straightforward manner, without complex interactions.

In summary, this program exhibits a mix of cohesion and coupling levels. Functions like input, display, and combine show high cohesion, as they perform well-defined and specific tasks. The coupling between functions is generally low, indicating that they interact with each other in a controlled and limited way. However, main and sort exhibit moderate cohesion due to their dual roles in controlling program flow and sorting, and they have moderate coupling due to their interaction with each other.

Algorithmic Metrics

Psuedocode:

```
sort(array)
   size = array.length
   source = array
   destination = new array with length of source
   number = 2

   while number > 1
      number = 0
      begin1 = 0

      while begin1 < size
         end1 = begin1 + 1
         while end1 < size and source[end1 - 1] =< source[end1]
            end1 ++

         begin2 = end1
         if begin2 < size
            end2 = begin2 + 1
         else
            end2 = begin2
         while end2 < size and source[end2 - 1] =< source[end2]
            end2 ++

         number ++
         combine(source, destination, begin1, begin2, end2)
         begin1 = end2

      hold = source
      source = destination
      destination = hold
   return source

combine(source, destination, begin1, begin2, end2)
   end1 = begin2

   for i = begin1 ... end2
      if (begin1 < end1) and (begin2 = end2 or source[begin1] < source[begin2])
```

```
        destination[i] = source[begin1]
        begin1 ++
      else
        destination[i] = source[begin2]
        begin2 ++
  return destination
```

Response

size = array.length: This line calculates the size of the input array. This operation takes O(1) time.

source = array and destination = new array with length of source: These lines create two arrays, source and destination, each with the same size as the input array. Copying the array and creating a new array both take O(n) time, where n is the size of the array.

number = 2: This is a constant assignment and takes O(1) time.

The outer while loop: while number > 1: This loop executes until number becomes 1. In each iteration, it performs a series of nested operations.

a. number = 0: This line assigns 0 to number. It's a constant time operation.

b. begin1 = 0: Similarly, this is a constant time operation.

c. The first inner while loop: while begin1 < size: This loop iterates through the entire array. The worst-case scenario is when the array is fully unsorted, and in that case, it will perform approximately n iterations (where n is the size of the array).

d. The second inner while loop: while end1 < size and source[end1 - 1] <= source[end1]: This loop searches for the end of the first subarray in a segment of the array. It may execute a number of times in the worst-case proportional to the size of the subarray, but in the overall picture, it contributes to O(n) time complexity because it's nested inside the first loop.

e. Similar operations are performed for begin2, end2, and the second inner while loop, all contributing to O(n) time complexity.

f. number++: This operation is a constant time operation.

g. combine(source, destination, begin1, begin2, end2): Calls a function combine. This operation depends on the time complexity of the combine function.

h. begin1 = end2: A constant time operation.

i. Swapping source and destination using a temporary variable hold takes O(n) time because it involves copying the entire array.

j. return source: A constant time operation.

The combine function:

a. The end1 = begin2 operation is a constant time operation.

b. The for loop inside the combine function: for i = begin1 ... end2: This loop iterates over the elements within the range specified. The range size depends on the input and can be up to n in the worst case.

c. Inside the loop, there are conditional statements and assignments that operate in constant time.

return destination in the combine function is a constant time operation.

In summary, the time complexity of this sorting code is O(n^2), where n is the size of the input array. The algorithm performs nested loops, and each iteration contributes to the overall time complexity. There are faster ways of doing this though.

Test Cases

Test Case	Input	Expected Output
Test Case 1	[5, 2, 8, 1, 9]	[1, 2, 5, 8, 9]
Test Case 2	[]	[]
Test Case 3	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5]
Test Case 4	[3, 1, 2, 2, 3]	[1, 2, 2, 3, 3]

Trace Verification

Step	Function	Input State	Output State
1	`main`	Input: [5, 2, 8, 1, 9]	-
2	`main`	Calls `sort`	-
2a	`sort`	Input: [5, 2, 8, 1, 9], Destination: Array of the same size, Begin1: 0, Begin2: 1, End2: 1	-
2a-i	`combine`	Merging [5] and [2, 8, 1, 9]	Result: [2, 5, 8, 1, 9]
2a-ii	`sort`	Continue sorting with [2, 5, 8, 1, 9], Begin1: 1, Begin2: 4, End2: 4	-
2a-iii	`combine`	Merging [2, 5, 8] and [1, 9]	Result: [1, 2, 5, 8, 9]
2a-iv	`sort`	Sorting complete, return sorted array [1, 2, 5, 8, 9] to `main`	-
3	`main`	Receives sorted array [1, 2, 5, 8, 9]	-
4	`display`	Display sorted array [1, 2, 5, 8, 9]	-