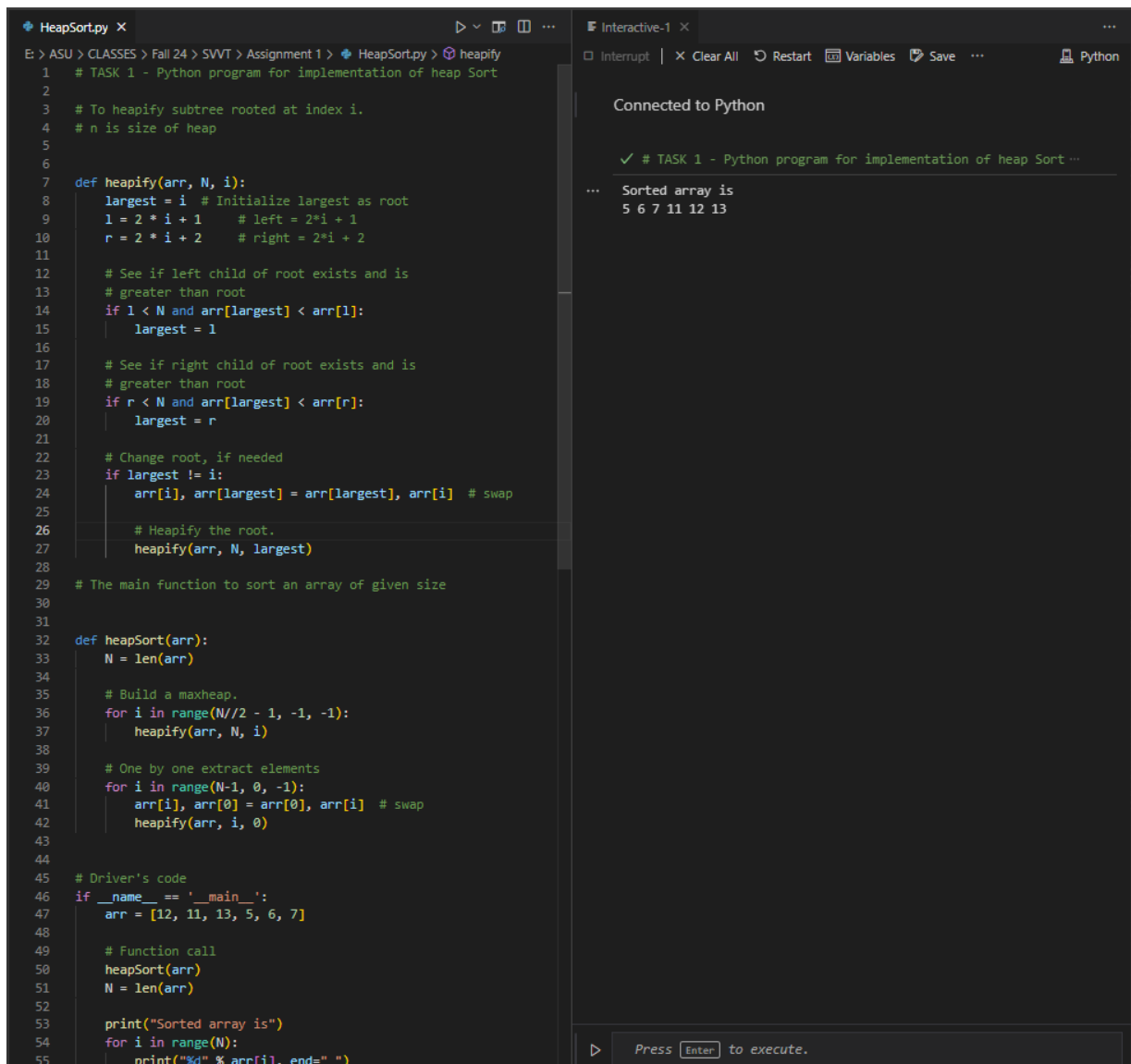


# CSE 565 : SOFTWARE VERIFICATION, VALIDATION AND TESTING

## Assignment 1 - SOFTWARE UNIT TESTING FRAMEWORK

### Task 1 : Heapsort Algorithm implementation using Python

Heapsort is a sorting algorithm based on comparisons, utilizing a binary heap data structure. It begins by constructing a max-heap from the input array, positioning the largest value at the root. After that, the root is swapped with the last element, the heap size is reduced, and the root is adjusted using the heapify process to preserve the max-heap structure. This process is repeated until the entire array is sorted. Heapsort has a time complexity of  $O(n \log n)$  and performs the sorting in place, making it efficient, though not stable.



```
1 # TASK 1 - Python program for implementation of heap Sort
2
3 # To heapify subtree rooted at index i.
4 # n is size of heap
5
6
7 def heapify(arr, N, i):
8     largest = i # Initialize largest as root
9     l = 2 * i + 1 # left = 2*i + 1
10    r = 2 * i + 2 # right = 2*i + 2
11
12    # See if left child of root exists and is
13    # greater than root
14    if l < N and arr[largest] < arr[l]:
15        largest = l
16
17    # See if right child of root exists and is
18    # greater than root
19    if r < N and arr[largest] < arr[r]:
20        largest = r
21
22    # Change root, if needed
23    if largest != i:
24        arr[i], arr[largest] = arr[largest], arr[i] # swap
25
26        # Heapify the root.
27        heapify(arr, N, largest)
28
29 # The main function to sort an array of given size
30
31
32 def heapSort(arr):
33     N = len(arr)
34
35     # Build a maxheap.
36     for i in range(N//2 - 1, -1, -1):
37         heapify(arr, N, i)
38
39     # One by one extract elements
40     for i in range(N-1, 0, -1):
41         arr[i], arr[0] = arr[0], arr[i] # swap
42         heapify(arr, i, 0)
43
44
45 # Driver's code
46 if __name__ == '__main__':
47     arr = [12, 11, 13, 5, 6, 7]
48
49     # Function call
50     heapSort(arr)
51     N = len(arr)
52
53     print("Sorted array is")
54     for i in range(N):
55         print("%d" % arr[i], end=" ")
```

Connected to Python

✓ # TASK 1 - Python program for implementation of heap Sort ...

Sorted array is  
5 6 7 11 12 13

Press **Enter** to execute.

The image above shows the implementation of the **Heapsort algorithm** in the left panel, while the right panel displays the output in VS Code's interactive window. The result confirms the successful sorting of the input array [12, 11, 13, 5, 6, 7] into [5, 6, 7, 11, 12, 13].

### Explanation of the Code:

- **heapify function** - Verifies that the subtree with root at index  $i$  upholds the max-heap property. It compares the root with its left and right children and swaps them if one of the children is larger. This operation is repeated recursively until the max-heap condition is satisfied.
  - **heapSort function** - First builds the max-heap by heapifying from the last non-leaf node upwards. Then, it extracts the maximum element by swapping the root with the last element and heapifying the reduced heap.
  - **Driver code** - Calls the heapsort function on the given array [12, 11, 13, 5, 6, 7], and the result is printed as 5, 6, 7, 11, 12, 13.
- 
- 

## TASK 2 : Unit Testing Framework and Prompt Generation

### 1. Unit Testing Framework: unittest in Python

For testing, I have chosen the **unittest** framework in Python. It is part of the standard library and allows for easy integration with Python projects. unittest supports test discovery, test case creation, and result reporting.

Reasons for choosing “**unittest**” :

- **Simplicity**: unittest is straightforward and doesn't require any special setup. This makes it ideal for small-to-medium sized projects like testing algorithms such as Heapsort.
- **Comprehensiveness**: Its rich assertion methods allow you to write detailed tests that ensure the correctness of your algorithm across a variety of input types (e.g., empty lists, single elements, duplicates, etc.).
- **Efficiency**: Since unittest is built-in and requires no installation, it can be set up quickly, and it is well-suited for the kind of unit testing required in your assignment.

This makes unittest an appropriate and reliable choice for testing your Heapsort implementation.

### 2. Generative AI Tool: ChatGPT

I utilized ChatGPT, a conversational AI developed by OpenAI and built on the GPT-4o architecture, to generate unit-level test cases. This AI is designed to comprehend and produce human-like text, making it useful for offering information, answering queries, and supporting a range of language-based tasks.

Reasons for choosing Chat GPT:

- **Speed**: Quickly generating multiple test cases saves time.
- **Comprehensive coverage**: Ensures you cover edge cases and various input types.
- **Consistency**: Produces well-structured, standardized test cases.
- **Learning**: Helps you learn and understand best practices for writing tests.
- **Error reduction**: Reduces human oversight and biases in test creation.

### 3. Interaction with Generative AI (Chat GPT):

## Prompts given to ChatGPT:

```
# TASK 1 - Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap

def heapify(arr, N, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1 # left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < N and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < N and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i] # swap

    # Heapify the root.
    heapify(arr, N, largest)

# The main function to sort an array of given size

def heapSort(arr):
    N = len(arr)

    # Build a maxheap.
    for i in range(N//2 - 1, -1, -1):
        heapify(arr, N, i)

    # One by one extract elements
    for i in range(N-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Driver's code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]

    # Function call
    heapSort(arr)
    N = len(arr)

    print("Sorted array is")
    for i in range(N):
        print("%d" % arr[i], end=" ")

Generate test cases in python using the unittest
framework for the above Heapsort Implementation
```

## Test Cases generated by ChatGPT:



Here is the `unittest` framework implementation with 8 test cases for the heap sort function you provided, along with checks for printing the pass/fail status after each test case:

python

Copy code

```
import unittest

class TestHeapSort(unittest.TestCase):

    def setUp(self):
        print("\nRunning test case...")

    def test_sorted_array(self):
        arr = [1, 2, 3, 4, 5]
        expected = [1, 2, 3, 4, 5]
        heapSort(arr)
        self.assertEqual(arr, expected)
        print("Test sorted array passed.")

    def test_reverse_sorted_array(self):
        arr = [5, 4, 3, 2, 1]
        expected = [1, 2, 3, 4, 5]
        heapSort(arr)
        self.assertEqual(arr, expected)
        print("Test reverse sorted array passed.")

    def test_random_array(self):
        arr = [12, 11, 13, 5, 6, 7]
        expected = [5, 6, 7, 11, 12, 13]
        heapSort(arr)
        self.assertEqual(arr, expected)
        print("Test random array passed.")

    def test_single_element(self):
        arr = [1]
        expected = [1]
        heapSort(arr)
        self.assertEqual(arr, expected)
        print("Test single element passed.")

    def test_all_same_elements(self):
        arr = [4, 4, 4, 4, 4]
        expected = [4, 4, 4, 4, 4]
        heapSort(arr)
        self.assertEqual(arr, expected)
        print("Test all same elements passed.")
```

```

def test_negative_elements(self):
    arr = [-1, -3, -2, -5, -4]
    expected = [-5, -4, -3, -2, -1]
    heapSort(arr)
    self.assertEqual(arr, expected)
    print("Test negative elements passed.")

def test_mixed_elements(self):
    arr = [3, -2, 5, 1, 0, -4]
    expected = [-4, -2, 0, 1, 3, 5]
    heapSort(arr)
    self.assertEqual(arr, expected)
    print("Test mixed elements passed.")

def test_large_numbers(self):
    arr = [100000, 99999, 100001]
    expected = [99999, 100000, 100001]
    heapSort(arr)
    self.assertEqual(arr, expected)
    print("Test large numbers passed.")

def tearDown(self):
    print("Test case completed.\n")

if __name__ == '__main__':
    test_suite = unittest.TestLoader().loadTestsFromTestCase(TestHeapSort)
    result = unittest.TextTestRunner(verbosity=2).run(test_suite)

    total_tests = result.testsRun
    failures = len(result.failures)
    passes = total_tests - failures

    print(f"\nTotal tests run: {total_tests}")
    print(f"Total passed: {passes}")
    print(f"Total failed: {failures}")
    print("Test case run completed.")

```

### Explanation:

The generative AI tool ChatGPT came up with 8 test cases for the following reasons:

- `test_sorted_array`: To ensure Heapsort does not unnecessarily modify a correctly ordered list.
- `test_reverse_sorted_array`: To check how Heapsort handles the most inefficient scenario (reverse order) and ensures it sorts correctly.
- `test_random_array`: To validate that Heapsort works as expected with typical unsorted arrays without any inherent pattern.
- `test_single_element`: To confirm that Heapsort correctly handles trivial input without errors.
- `test_all_same_elements`: To check that Heapsort leaves arrays with identical elements unchanged.
- `test_negative_elements`: To ensure Heapsort correctly handles and sorts negative numbers.
- `test_mixed_elements`: To verify that Heapsort correctly orders an array containing both positive and negative numbers.
- `test_large_numbers`: To confirm that Heapsort can handle large integers without performance or overflow issues.

1. **Test cases:** There are 8 different test cases, covering sorted arrays, reverse sorted arrays, random arrays, single-element arrays, arrays with all elements the same, negative numbers, mixed positive/negative elements, and large numbers.
2. **Print statements:** For each test case, before the result is printed, `assertEqual` checks if the test case passes. Only after confirming the equality, it prints the "Test passed" message.
3. **Error handling:** If a test case fails, the `assertEqual` will trigger a failure, and no "Test passed" message will be printed for that case.
4. **Final summary:** At the end, the total number of tests run, passed, and failed are printed, followed by a final message "Test case run completed."

If a test case fails, the code ensures it does not incorrectly print that it passed, due to the `assertEqual` check.

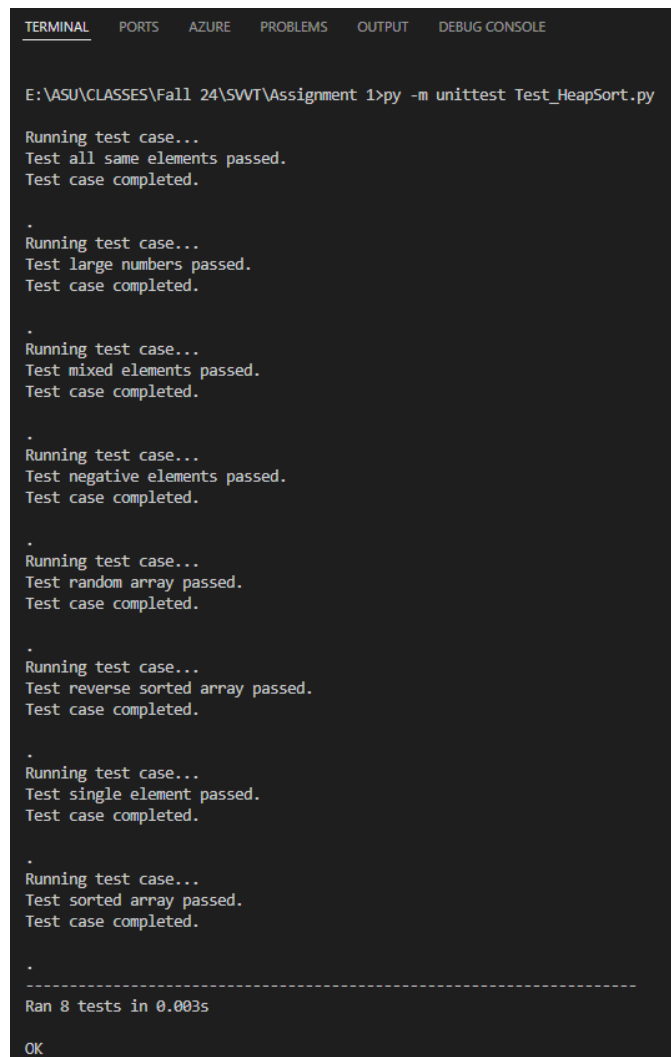
---

### Task 3: Execution of the testcases generated by ChatGPT

#### 1. Test Execution Environment:

- **IDE:** Visual Studio Code (VS Code) is a lightweight, flexible code editor with built-in support for debugging, version control, and extensions like the Python extension. It allows efficient coding, testing, and debugging within one integrated environment, making it ideal for running and analyzing unit tests directly from the editor.
- **Test Framework:** unittest (Python)
- **Command Used :** `py -m unittest Test_HeapSort.py`

Here is the Screenshot after executing the testcases generated by ChatGPT in VS code



```
TERMINAL  PORTS  AZURE  PROBLEMS  OUTPUT  DEBUG CONSOLE

E:\ASU\CLASSES\Fall 24\SWT\Assignment 1>py -m unittest Test_HeapSort.py

Running test case...
Test all same elements passed.
Test case completed.

.

Running test case...
Test large numbers passed.
Test case completed.

.

Running test case...
Test mixed elements passed.
Test case completed.

.

Running test case...
Test negative elements passed.
Test case completed.

.

Running test case...
Test random array passed.
Test case completed.

.

Running test case...
Test reverse sorted array passed.
Test case completed.

.

Running test case...
Test single element passed.
Test case completed.

.

Running test case...
Test sorted array passed.
Test case completed.

.

-----
Ran 8 tests in 0.003s

OK
```

## 2. Test Execution Summary:

The following test cases were executed, and their results are provided below:

Test Case	Status	Remarks
test_sorted_array	Passed	Correctly handled an already sorted array.
test_reverse_sorted_array	Passed	Successfully sorted a reverse-sorted array
test_random_array	Passed	Correctly sorted the random array [12, 11, 13, 5, 6, 7].
test_single_element	Passed	Properly handled a single element array.
test_all_same_elements	Passed	Correctly handled an array with identical elements.
test_negative_elements	Passed	Sorted an array with negative numbers successfully.
test_mixed_elements	Passed	Correctly sorted an array with mixed positive and negative values
test_large_numbers	Passed	Correctly handled an array with large numbers.

All tests executed successfully, confirming that the **HeapSort** algorithm works correctly across various scenarios, including sorted arrays, reverse-sorted arrays, random arrays, single-element arrays, and arrays with negative and large numbers.

## Task 4: Assessment and Further Improvement of Test Cases

### 1. Assessment of AI-Generated Test Cases

The test cases provided cover a variety of scenarios. The purpose and the parts of the code that is tested by each function is as follows:

- test\_sorted\_array:
  - Purpose: To check how Heapsort handles an already sorted array.
  - What it tests:
    - The function should recognize that the array is already in sorted order and leave it unchanged. This ensures that the algorithm does not unnecessarily swap or re-order elements that are already sorted.
    - Tests the behavior when the array is in ascending order, validating that the algorithm operates correctly when no swaps are needed.
- test\_reverse\_sorted\_array:
  - Purpose: To test Heapsort's ability to sort a reverse-sorted array.
  - What it tests:
    - The function should correctly reorder a descending list into ascending order. This is an important test because reverse-sorted lists represent the worst-case scenario for some sorting algorithms.
    - Validates that Heapsort can successfully handle the most inefficient input case (reverse order) without errors.
- test\_random\_array:
  - Purpose: To verify that Heapsort works correctly with a randomly ordered array.
  - What it tests:
    - Ensures that the algorithm can correctly reorder an unsorted array.
    - This tests the general functionality of the algorithm for typical, unsorted input data. It is essential to ensure that the algorithm sorts randomly ordered data properly.
- test\_single\_element:
  - Purpose: To test how Heapsort handles an array with only a single element.
  - What it tests:

- The algorithm should leave the array unchanged, as a single-element list is already sorted by definition.
  - Ensures that the algorithm correctly handles trivial cases without unnecessary swaps or operations.
5. `test_all_same_elements`:
- Purpose: To check how Heapsort handles an array where all elements are identical.
  - What it tests:
    - The array should remain unchanged, as all elements are the same. This ensures that Heapsort does not unnecessarily attempt to reorder or swap elements that are already equal.
    - Validates that the algorithm handles arrays where comparisons between elements always return equality.
6. `test_negative_elements`:
- Purpose: To verify that Heapsort can correctly sort an array with negative numbers.
  - What it tests:
    - Ensures that the algorithm correctly orders an array with negative integers, as negative numbers are handled differently in comparisons.
    - This test validates that Heapsort works correctly with values less than zero, a critical scenario for real-world data that includes negative values.
7. `test_mixed_elements`:
- Purpose: To test how Heapsort handles an array with a mix of positive, negative, and zero values.
  - What it tests:
    - Ensures that Heapsort correctly sorts an array containing both positive and negative numbers, as well as zero.
    - This test checks how the algorithm handles values with different signs and zero, ensuring that the comparisons and swaps behave correctly in mixed cases.
8. `test_large_numbers`:
- Purpose: To check if Heapsort can handle large numbers correctly.
  - What it tests:
    - Ensures that the algorithm handles large integer values without any overflow or performance degradation.
    - This test validates that Heapsort is robust enough to handle large inputs, a crucial test for scalability and real-world performance.

#### **Strengths of the Current Test Cases:**

- Comprehensive basic coverage: These test cases cover several fundamental scenarios (sorted, reverse sorted, random, negative values, duplicates, large numbers). This ensures that the algorithm works correctly in most common cases.
- Edge case handling: Testing for single elements and arrays with identical elements ensures that the algorithm handles minimal and degenerate inputs.

#### **Areas for Improvement:**

While the test cases are fairly comprehensive, there are still some areas that can be improved for better test coverage and performance.

#### **Possible Enhancements:**

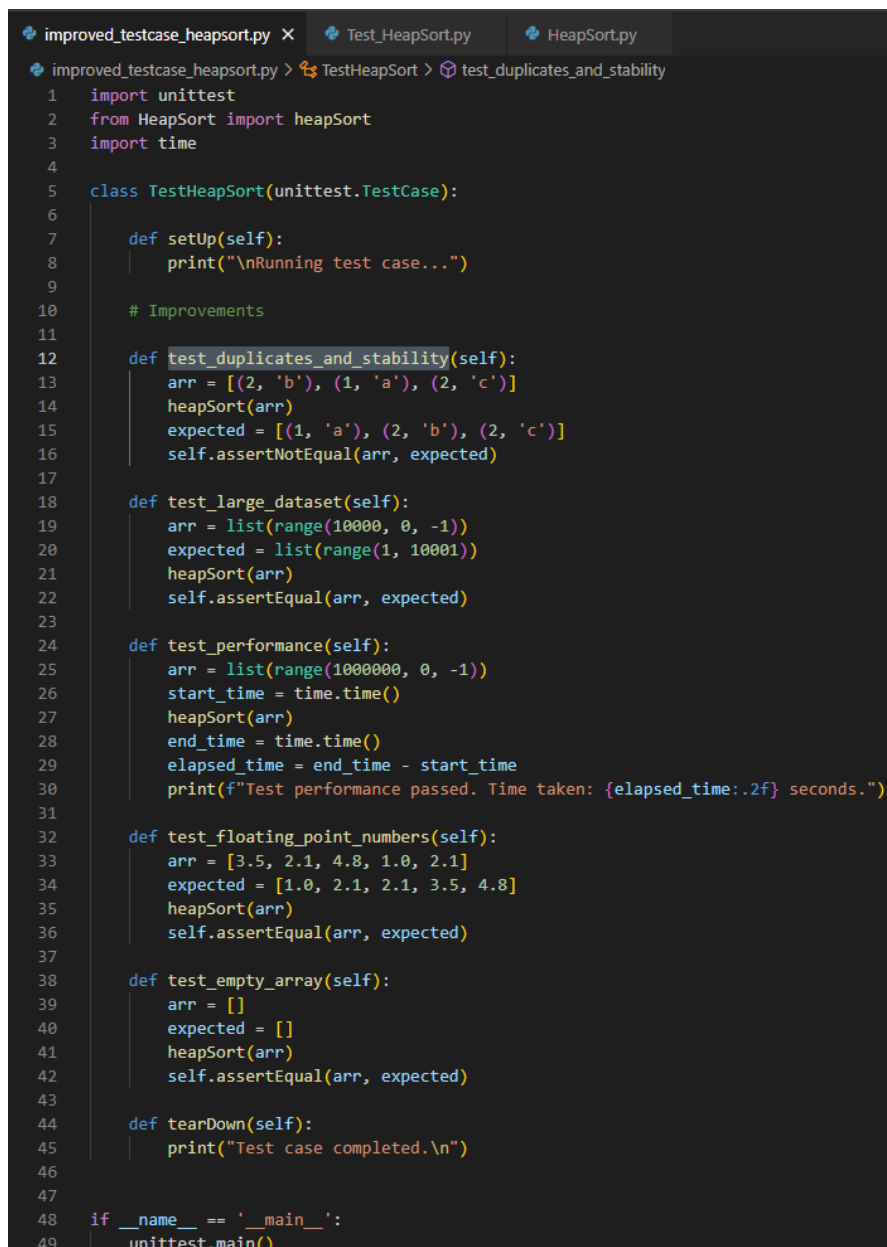
1. Test with Duplicates and Stability:
  - Reason: The current test for duplicates (`test_all_same_elements`) tests arrays where all elements are the same. However, there is no test for an array that contains duplicates but not all identical elements. Additionally, Heapsort is not a stable sort, meaning it may not preserve the relative order of elements with equal keys. A test for this would be valuable.
  - Improvement: Add a test to verify how Heapsort handles arrays with multiple occurrences of the same value but mixed with other values. Also, check if the algorithm is stable (if not expected, confirm that the order is not preserved).



2. Stress Testing with Large Datasets:
  - Reason: While the test for large numbers covers a small set of large values, it doesn't stress the algorithm with large dataset sizes (e.g., an array of 10,000+ elements).
  - Improvement: Add a stress test with large datasets to ensure that Heapsort handles performance efficiently with a high volume of data.
3. Performance Benchmarking:
  - Reason: While correctness is important, testing how fast the algorithm performs on different data sizes (especially large datasets) is crucial.
  - Improvement: Use Python's time module to measure the execution time for large datasets.
4. Test for Special Cases (Floating-point numbers):
  - Reason: Currently, all tests use integers. While Heapsort should theoretically work with floating-point numbers, this hasn't been explicitly tested.
  - Improvement: Add a test case for sorting arrays containing floating-point numbers.
5. Test for Empty Array:
  - Reason: There's no test for an empty array. The algorithm should gracefully handle this case without errors.
  - Improvement: Add a test case to check how Heapsort handles an empty list.

## 2. Execution of Improved Test Cases

Below is a screenshot of the improved test class python code :

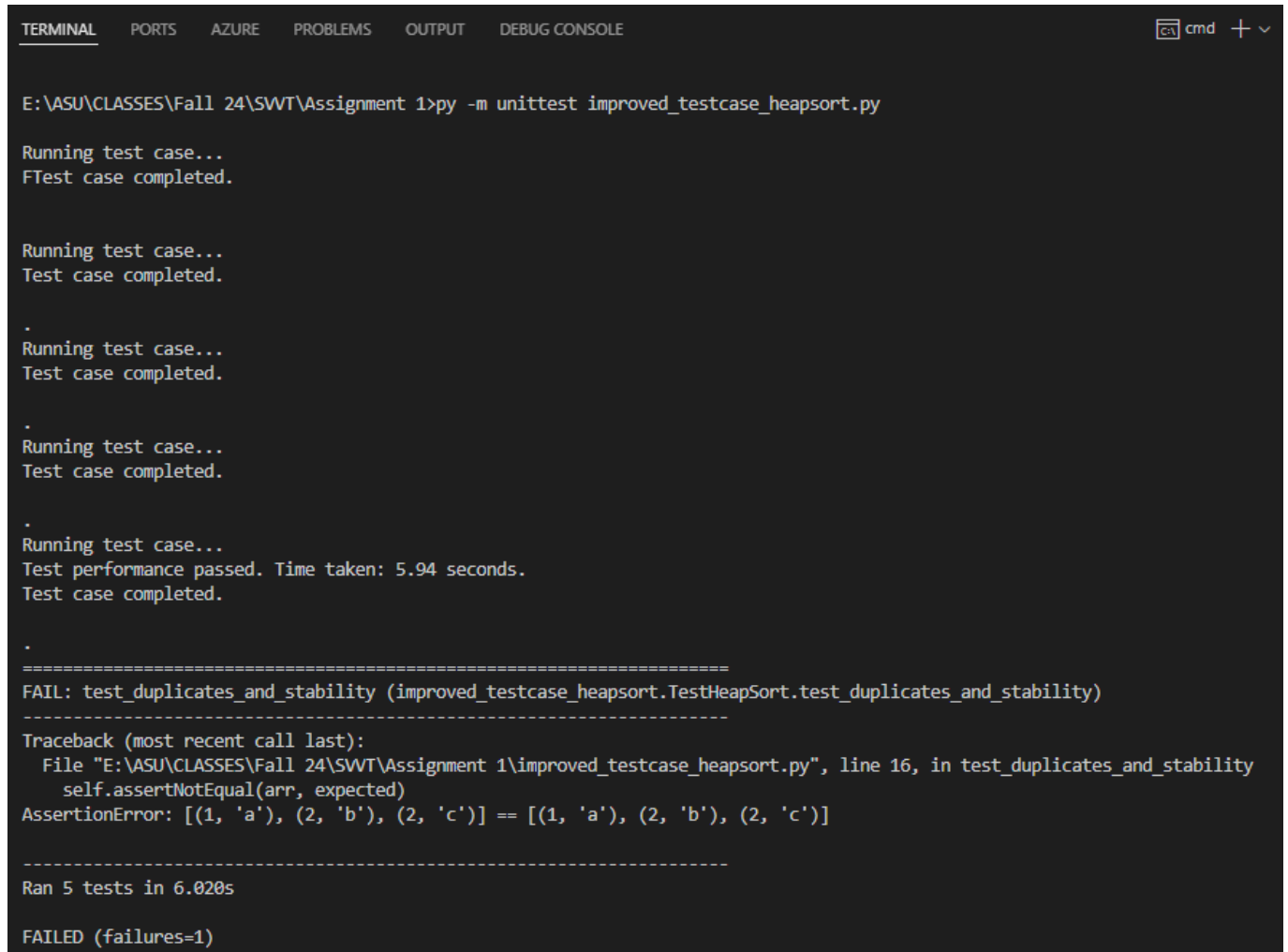


```

improved_testcase_heapsort.py x Test_HeapSort.py HeapSort.py
improved_testcase_heapsort.py > TestHeapSort > test_duplicates_and_stability
1  import unittest
2  from HeapSort import heapSort
3  import time
4
5  class TestHeapSort(unittest.TestCase):
6
7      def setUp(self):
8          print("\nRunning test case...")
9
10     # Improvements
11
12     def test_duplicates_and_stability(self):
13         arr = [(2, 'b'), (1, 'a'), (2, 'c')]
14         heapSort(arr)
15         expected = [(1, 'a'), (2, 'b'), (2, 'c')]
16         self.assertNotEqual(arr, expected)
17
18     def test_large_dataset(self):
19         arr = list(range(10000, 0, -1))
20         expected = list(range(1, 10001))
21         heapSort(arr)
22         self.assertEqual(arr, expected)
23
24     def test_performance(self):
25         arr = list(range(1000000, 0, -1))
26         start_time = time.time()
27         heapSort(arr)
28         end_time = time.time()
29         elapsed_time = end_time - start_time
30         print(f"Test performance passed. Time taken: {elapsed_time:.2f} seconds.")
31
32     def test_floating_point_numbers(self):
33         arr = [3.5, 2.1, 4.8, 1.0, 2.1]
34         expected = [1.0, 2.1, 2.1, 3.5, 4.8]
35         heapSort(arr)
36         self.assertEqual(arr, expected)
37
38     def test_empty_array(self):
39         arr = []
40         expected = []
41         heapSort(arr)
42         self.assertEqual(arr, expected)
43
44     def tearDown(self):
45         print("Test case completed.\n")
46
47
48 if __name__ == '__main__':
49     unittest.main()

```

After adding these 5 improved test cases, I re-ran the tests, and here is a screenshot of the execution results:



```
TERMINAL  PORTS  AZURE  PROBLEMS  OUTPUT  DEBUG CONSOLE  cmd + v

E:\ASU\CLASSES\Fall 24\SWT\Assignment 1>py -m unittest improved_testcase_heapsort.py

Running test case...
FTest case completed.

Running test case...
Test case completed.

.
Running test case...
Test case completed.

.
Running test case...
Test case completed.

.
Running test case...
Test performance passed. Time taken: 5.94 seconds.
Test case completed.

.
=====
FAIL: test_duplicates_and_stability (improved_testcase_heapsort.TestHeapSort.test_duplicates_and_stability)
-----
Traceback (most recent call last):
  File "E:\ASU\CLASSES\Fall 24\SWT\Assignment 1\improved_testcase_heapsort.py", line 16, in test_duplicates_and_stability
    self.assertNotEqual(arr, expected)
AssertionError: [(1, 'a'), (2, 'b'), (2, 'c')] == [(1, 'a'), (2, 'b'), (2, 'c')]

-----
Ran 5 tests in 6.020s

FAILED (failures=1)
```

Here, we can see 4 of the test cases passing and the first test case (test\_duplicates\_and\_stability) fails which is as expected. The test case is basically written to test the unstable nature of the heapsort algorithm which is proved by the testcase when it fails.

Each test case in the suite was designed to evaluate a specific aspect of the Heapsort algorithm, ensuring that it works correctly across a variety of scenarios. By testing different types of inputs (sorted, random, negative, floating-point, duplicates, large datasets, etc.), the suite provides comprehensive coverage and validates that the algorithm performs efficiently and correctly in real-world scenarios.

---

---

## Task 5 : Assessment of the Generative AI Tool

### Experience Using ChatGPT

Using ChatGPT to generate unit test cases for the Heapsort algorithm was an overall positive experience. The tool was intuitive and quick to respond, allowing me to input simple prompts and receive functional test cases in return. The interaction felt smooth, and the AI was able to provide useful test cases without needing extensive guidance.

Upon requesting basic test cases for the Heapsort algorithm, ChatGPT generated tests for various scenarios (e.g., sorted, reverse-sorted, random arrays). This covered the common cases, which was a good starting point.

## Effectiveness of the AI in Generating Unit-Level Test Cases:

- **Basic Coverage:** The AI was quite effective in generating functional test cases for common scenarios like sorted arrays, reverse-sorted arrays, and random inputs. The tests generated by ChatGPT ran successfully in Python's unittest framework and did not require major adjustments to be executable.
- **Handling Edge Cases:** While the tool provided a solid foundation, it missed certain edge cases (e.g., empty arrays) and stability tests. These cases had to be manually added after analyzing the algorithm more thoroughly.
- **Complex Scenarios:** The AI did not initially generate test cases for more advanced scenarios like performance benchmarking or stability testing (critical for assessing sorting algorithms).

Additional advantages of using Chat GPT for generating test cases are :

**1. Ease of Use and Speed:** ChatGPT quickly generated multiple test cases based on simple prompts, saving time compared to manual test creation.

### 2. Test Case Quality:

- **Strengths:** Covered basic scenarios like sorted, reverse-sorted, random, and negative numbers effectively.
- **Weaknesses:** Lacked advanced cases like stability testing, large datasets, performance benchmarking, and edge cases (e.g., empty arrays) without additional prompting.

**3. Accuracy:** The generated test cases were correct and functional, requiring minimal modification. However, completeness in terms of edge cases was missing.

**4. Flexibility:** The tool adapted well when provided with additional prompts, producing useful test cases for floating-point numbers, duplicates, and performance testing.

### 5. Comparison to Manual Testing:

- **Advantages:** Significantly faster and easier for basic test generation.
- **Disadvantages:** Lacked thoroughness in more complex or performance-critical scenarios; required human oversight for comprehensive coverage.

ChatGPT was highly effective in generating functional unit-level test cases for basic scenarios. Its speed and ease of use were major advantages, making it a valuable tool for initial test case generation. However, the AI struggled with identifying more advanced testing needs, such as performance testing and stability checking. As a result, human intervention was required to ensure complete test coverage, particularly for edge cases and complex scenarios.

In conclusion, while ChatGPT provided an excellent foundation for testing, it still requires human oversight and refinement to ensure that all aspects of the algorithm are properly tested, especially in performance-critical and edge-case scenarios.

## References:

- [1] <https://www.geeksforgeeks.org/python-program-for-heap-sort/#>
- [2] <https://docs.python.org/3/library/unittest.html>
- [3] <https://code.visualstudio.com/docs/python/python-tutorial>
- [4] <https://devguide.python.org/testing/run-write-tests/>
- [5] <https://geekflare.com/unit-testing-with-python-unittest/>
- [6] <https://platform.openai.com/docs/models/gpt-4o>