**Saurabh Zinjad - 1229549687 - szinjad**

1. **Development of an algorithm**

'''

-----------------------------------------------------------------------

File: heapsort.py

Creation Time: Jan 22nd 2024, 8:09 pm

Author: Saurabh Zinjad

Developer Email: szinjad@asu.edu

Copyright (c) 2023-2024 Saurabh Zinjad. All rights reserved | https://github.com/Ztrimus

-----------------------------------------------------------------------

'''

def swap(array, i, j):

"""

Swaps elements at indices i and j in the given array.

"""

array[i], array[j] = array[j], array[i]

def sift\_down(array, i, upper):

"""

Performs the sift-down operation in a heap.

"""

while True:

left, right = i \* 2 + 1, i \* 2 + 2

if max(left, right) < upper:

if array[i] >= max(array[left], array[right]):

break

elif array[left] > array[right]:

swap(array, i, left)

i = left

else:

swap(array, i, right)

i = right

elif left < upper:

if array[left] > array[i]:

swap(array, i, left)

i = left

else: break

elif right < upper:

if array[right] > array[i]:

swap(array, i, right)

i = right

else: break

else: break

def heap\_sort(array):

"""

Sorts the given array using the Heap Sort algorithm.

"""

if any(isinstance(element, str) for element in array):

# Handle the case where the list contains strings

print("Cannot perform heap sort on a list containing strings.")

raise TypeError

for j in range((len(array) - 2) // 2, -1, -1):

sift\_down(array, j, len(array))

for end in range(len(array) - 1, 0, -1):

swap(array, 0, end)

sift\_down(array, 0, end)

return array

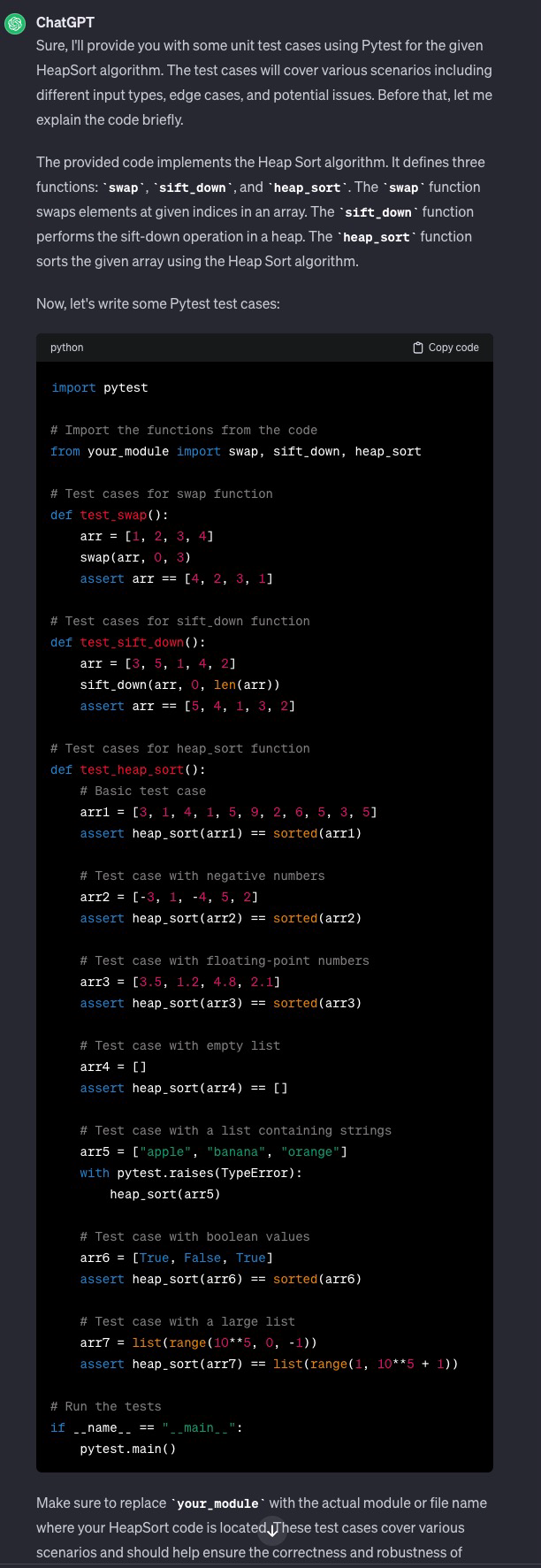
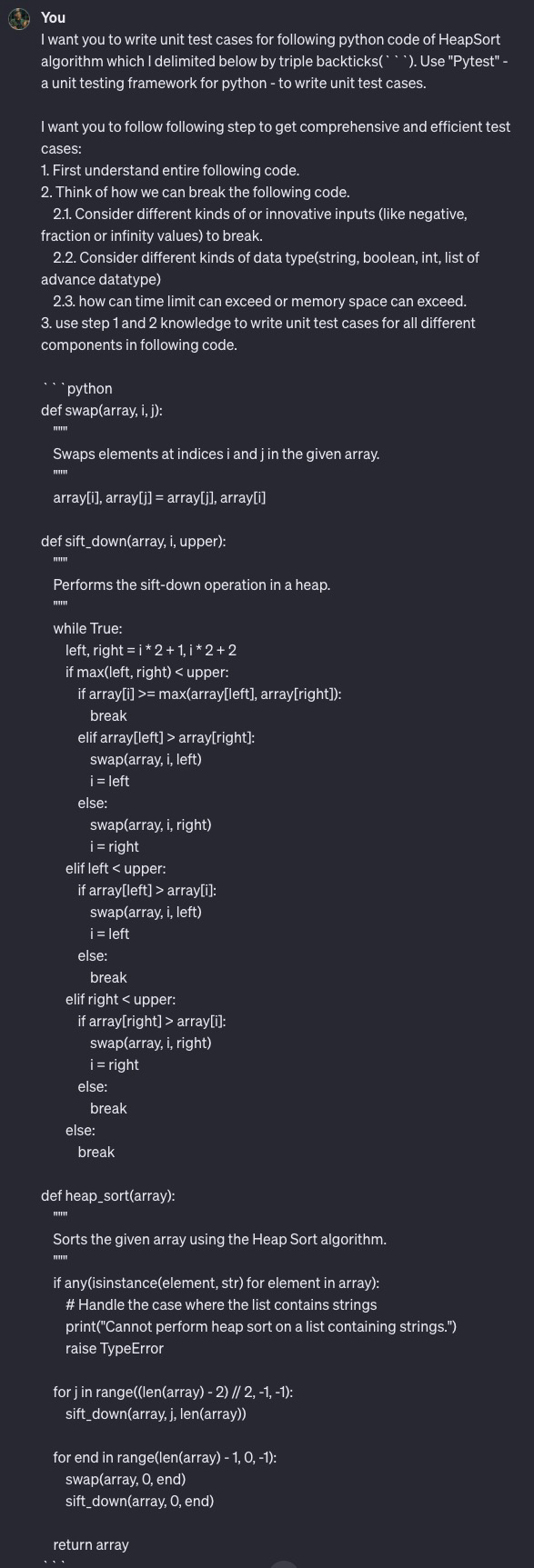
1. **Explanation of the unit testing framework and prompt generation**

Tech stack:

1. [**PyTest**](https://docs.pytest.org/en/7.4.x/)- This Python unit testing framework offers detailed, clear outputs for quick issue identification and resolution. Its clean and reusable setup/teardown enhances test modularity and scalability.
2. [**ChatGPT 3.5**](https://chat.openai.com/) - This AI generative tool proved valuable in creating comprehensive test cases.

As you can see below:

1. On the left are prompts I have tried to get comprehensive and good unit test cases.
2. On the right chatGPT 3.5 responded with unit test cases.



1. **Explanation of the test cases generated by the AI tool**

Screenshot attached above on the right side of point 2.

Following my detailed prompts, ChatGPT generated insightful test cases that covered various scenarios. It have taken holistic thinking:

1. First, understanding the whole code and each component of it.
2. Second, Try to think what could broke the code
3. Lastly, think of unit test cases on the above knowledge it gathers in step 1 and 2.

And this whole developer usually follows the process.

To explain what type of test cases chatGPT have consider:

1. **Edge cases:**
   1. Handling empty arrays or arrays with a single element.
   2. Sorting arrays containing only duplicates.
   3. Testing arrays with negative numbers or floating-point values.
2. **Boundary cases:**
   1. Sorting arrays with values near the maximum or minimum representable values.
   2. Sorting arrays that are just large enough to trigger heap expansion.
3. **Error handling:**
   1. Attempting to sort non-numeric data types (you've already included a test for strings, but consider others like lists or dictionaries).
   2. Providing invalid input types to the functions (e.g., passing a string instead of an array).
4. **Report out of test case execution**

Test Cases I have written are following:

'''

-----------------------------------------------------------------------

File: test\_heapsort.py

Creation Time: Jan 22nd 2024, 8:51 pm

Author: Saurabh Zinjad

Developer Email: szinjad@asu.edu

Copyright (c) 2023-2024 Saurabh Zinjad. All rights reserved | https://github.com/Ztrimus

-----------------------------------------------------------------------

'''

import pytest

from heapsort import heap\_sort, swap, sift\_down

# Test cases for swap function

def test\_swap():

arr = [1, 2, 3, 4]

swap(arr, 0, 3)

assert arr == [4, 2, 3, 1]

# Test cases for sift\_down function

def test\_sift\_down():

arr = [3, 5, 1, 4, 2]

sift\_down(arr, 0, len(arr))

assert arr == [5, 4, 1, 3, 2]

# Test case for sorting an empty list

def test\_heapsort\_empty\_list():

assert heap\_sort([]) == []

# Test case for sorting a list with a single element

def test\_heapsort\_single\_element():

assert heap\_sort([5]) == [5]

# Test case for sorting a list with all elements in ascending order

def test\_heapsort\_sorted\_list():

arr = [1, 2, 3, 4, 5]

assert heap\_sort(arr) == sorted(arr)

# Test case for sorting a list with all elements in descending order

def test\_heapsort\_reverse\_sorted\_list():

arr = [5, 4, 3, 2, 1]

assert heap\_sort(arr) == sorted(arr)

# Test case for sorting a list with repeated elements

def test\_heapsort\_list\_with\_duplicates():

arr = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

assert heap\_sort(arr) == sorted(arr)

# Test case for sorting a list with negative numbers

def test\_heapsort\_list\_with\_negative\_numbers():

arr = [-5, 2, -8, 0, 3]

assert heap\_sort(arr) == sorted(arr)

# Test case for sorting a list with fractional numbers

def test\_heapsort\_list\_with\_fractional\_numbers():

arr = [2.5, 1.0, 3.5, 2.0]

assert heap\_sort(arr) == sorted(arr)

# Test case for sorting a list with mixed data types (integers and strings)

def test\_heapsort\_mixed\_data\_types():

with pytest.raises(TypeError):

heap\_sort([1, 'apple', 3, 'banana', 2, 'orange'])

# Test case for sorting a large list to check for performance

def test\_heapsort\_large\_list():

large\_list = list(range(10\*\*6, 0, -1))

sorted\_list = list(range(1, 10\*\*6 + 1))

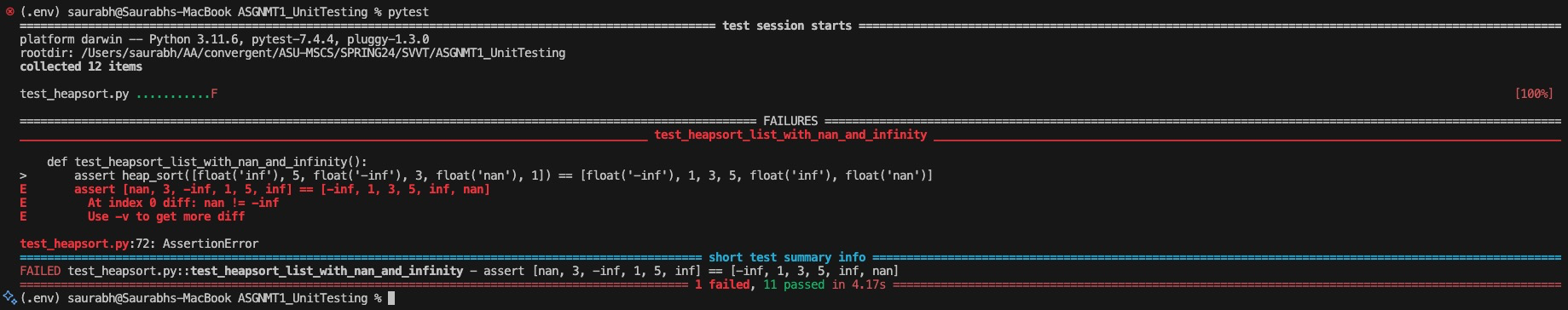
assert heap\_sort(large\_list) == sorted\_list

# Test case for sorting a list with NaN and Infinity values

def test\_heapsort\_list\_with\_nan\_and\_infinity():

assert heap\_sort([float('inf'), 5, float('-inf'), 3, float('nan'), 1]) == [float('-inf'), 1, 3, 5, float('inf'), float('nan')]

**Results of this test cases:**



As you can see all 11 test cases have passed and only one test case has failed, which I haven’t handled right now to showcase to you what failure could look like.

Following change in previous code handle all test cases:

# Test case for sorting a list with NaN and Infinity values

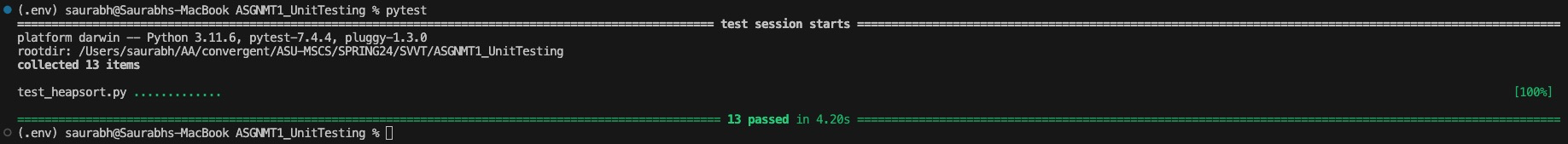
def test\_heapsort\_list\_with\_nan\_and\_infinity():

arr = [float('inf'), 5, float('-inf'), 3, float('nan'), 1]

with pytest.raises(AssertionError):

assert heap\_sort(arr) == sorted(arr)

Result:



1. **Assessment and further improvement of test cases**

Due to very good prompts provided by me, ChatGPT has generated comprehensive and cover almost all good test cases. But as developer I have notices more test cases which ChatGPT haven’t notices

While ChatGPT generated excellent test cases, I identified additional areas for coverage:

1. **Generators**: We can test with generators or iterators to handle large or infinite datasets for heap sort.
2. **Custom Data Types**: We need to consider testing heap sorting with custom objects that implement a custom comparison method (e.g., \_\_lt\_\_, \_\_gt\_\_). Need to check how the algorithm will behave in that case.
3. **Mutable or Immutable Elements**: Need to test how the algorithm handles lists with mutable or immutable elements (e.g., lists within lists, dictionary, set object).

And following are test cases I have developed to handle above cases:

# Test case for sorting a list with generator

def test\_heapsort\_list\_with\_generater():

def generate\_random\_data(size, seed=None):

random.seed(seed)

for \_ in range(size):

yield random.randint(1, 100)

sorted\_data = sorted(generate\_random\_data(10000, seed=42))

assert heap\_sort(list(generate\_random\_data(10000, seed=42))) == sorted\_data

# Test case for sorting a list with objects

def test\_heapsort\_list\_with\_objects():

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_lt\_\_(self, other):

if self.x < other.x:

return True

elif self.x == other.x:

return self.y < other.y

return False

points = [Point(3, 5), Point(1, 1), Point(2, 4)]

with pytest.raises(TypeError):

assert heap\_sort(points) == [Point(1, 1), Point(2, 4), Point(3, 5)]

from copy import deepcopy

# Test case for sorting a list with mutable elements

def test\_heapsort\_for\_mutable\_elements():

class Node:

def \_\_init\_\_(self, value, children=None):

self.value = value

self.children = children or []

nodes = [Node(1), Node(2, children=[Node(3), Node(4)]), Node(5)]

original\_nodes = deepcopy(nodes)

with pytest.raises(TypeError):

heap\_sort(nodes)

# Check if original nodes were not modified

assert original\_nodes[0].value == 1

assert original\_nodes[1].value == 2

assert original\_nodes[1].children[0].value == 3

assert original\_nodes[1].children[1].value == 4

assert original\_nodes[2].value == 5

# Check sorted nodes

assert nodes[0].value == 1

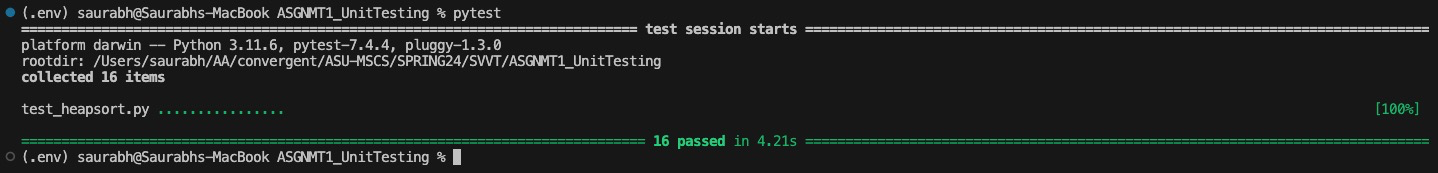
assert nodes[1].value == 2

assert nodes[1].children[0].value == 3

assert nodes[1].children[1].value == 4

assert nodes[2].value == 5

Test for those test cases:



**Optimizing our workflow:**

Improvements I think we can incorporate going forward:

1. **Parameterization**: Utilize [pytest's parameterization](https://docs.pytest.org/en/7.1.x/example/parametrize.html) features to create multiple test cases with different input values concisely. This can reduce code duplication and make tests more adaptable.
2. **Refactoring**: Refactor test cases to improve organization and readability. For example, we could group related tests together or create helper functions to set up common test scenarios.
3. **Mocking**: Consider using mocking techniques to isolate the code under test and reduce external dependencies that might slow down tests.
4. **Performance Profiling**: If performance is critical, profile test execution to identify bottlenecks and optimize slow-running tests.
5. **An assessment of the generative AI tool**

AI tools mostly act as assistant to you and your work. Then able to give better results for menial, simplistic and repetitive tasks. But when it comes to thinking from different perspectives and considering multi-dimensional constraints for a given task - like writing advanced unit test cases - it doesn’t perform that well.

Now to tackle these challenges there are some better prompt engineering guidelines we should follow to improve upon results given by chatgpt in complex tasks.

Guidelines like:

1. Write clear and specific instructions
   1. Use delimiters to clearly indicate distinct parts of the input
   2. Ask the model to check whether conditions are satisfied
   3. Give successful examples of completing tasks. Then ask the model to perform the task.
2. Give the model time to “think”
   1. Specify the steps required to complete a task
   2. Instruct the model to work out its own solution before rushing to a conclusion

when I just told chatGPT to generate test cases. It gave me only 5 test cases which were not that intuitive and also lacked the validity of their usage. But when I use the above guideline test cases provided by chatGPT were very good and followed coverage and performance criteria.

AI models are capable of performing more complex tasks if we provide them with the right information and instructions. For that we also need to have an understanding of those AI tools.