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
```

2. Explanation of the unit testing framework and prompt generation

Tech stack:

1. **PyTest** - 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** - 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.

You

I want you to write unit test cases for following python code of HeapSort algorithm which I delimited below by triple backticks(` ` `). Use "Pytest" - a unit testing framework for python - to write unit test cases.

I want you to follow following step to get comprehensive and efficient test cases:
1. First understand entire following code.
2. Think of how we can break the following code.
  2.1. Consider different kinds of or innovative inputs (like negative, fraction or infinity values) to break.
  2.2. Consider different kinds of data type(string, boolean, int, list of advance datatype)
  2.3. how can time limit can exceed or memory space can exceed.
3. use step 1 and 2 knowledge to write unit test cases for all different components in following code.

```python
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):
```

ChatGPT

Sure, I'll provide you with some unit test cases using Pytest for the given HeapSort algorithm. The test cases will cover various scenarios including different input types, edge cases, and potential issues. Before that, let me explain the code briefly.

The provided code implements the Heap Sort algorithm. It defines three functions: `swap`, `sift_down`, and `heap_sort`. The `swap` function swaps elements at given indices in an array. The `sift_down` function performs the sift-down operation in a heap. The `heap_sort` function sorts the given array using the Heap Sort algorithm.

Now, let's write some Pytest test cases:

python — Copy code

```python
import pytest

# Import the functions from the code
from your_module import swap, sift_down, heap_sort

# 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 cases for heap_sort function
def test_heap_sort():
    # Basic test case
    arr1 = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
    assert heap_sort(arr1) == sorted(arr1)

    # Test case with negative numbers
    arr2 = [-3, 1, -4, 5, 2]
    assert heap_sort(arr2) == sorted(arr2)

    # Test case with floating-point numbers
    arr3 = [3.5, 1.2, 4.8, 2.1]
    assert heap_sort(arr3) == sorted(arr3)

    # Test case with empty list
    arr4 = []
    assert heap_sort(arr4) == []

    # Test case with a list containing strings
    arr5 = ["apple", "banana", "orange"]
    with pytest.raises(TypeError):
        heap_sort(arr5)

    # Test case with boolean values
    arr6 = [True, False, True]
    assert heap_sort(arr6) == sorted(arr6)

    # Test case with a large list
    arr7 = list(range(10**5, 0, -1))
    assert heap_sort(arr7) == list(range(1, 10**5 + 1))

# Run the tests
if __name__ == "__main__":
    pytest.main()
```

3. 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:**
   a. Handling empty arrays or arrays with a single element.
   b. Sorting arrays containing only duplicates.
   c. Testing arrays with negative numbers or floating-point values.
2. **Boundary cases:**
   a. Sorting arrays with values near the maximum or minimum representable values.
   b. Sorting arrays that are just large enough to trigger heap expansion.
3. **Error handling:**
   a. Attempting to sort non-numeric data types (you've already included a test for strings, but consider others like lists or dictionaries).
   b. 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)
```

```python
# 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')]
```

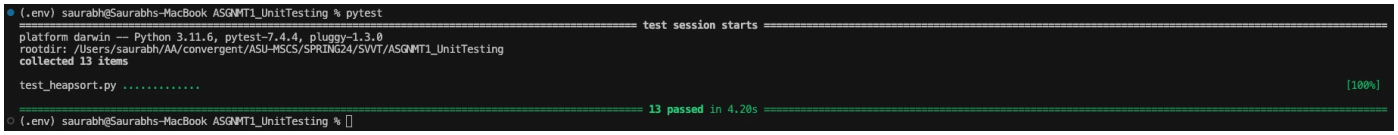
Results of this test cases:

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:

```python
# 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:



5. 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)]
```

```python
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:

```
(.env) saurabh@Saurabhs-MacBook ASGNMT1_UnitTesting % pytest
============================= test session starts ==============================
platform darwin -- Python 3.11.6, pytest-7.4.4, pluggy-1.3.0
rootdir: /Users/saurabh/AA/convergent/ASU-MSCS/SPRING24/SVVT/ASGNMT1_UnitTesting
collected 16 items

test_heapsort.py ................                                        [100%]

============================== 16 passed in 4.21s ==============================
(.env) saurabh@Saurabhs-MacBook ASGNMT1_UnitTesting %
```

Optimizing our workflow:

Improvements I think we can incorporate going forward:

1. **Parameterization**: Utilize pytest's parameterization 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.

6. 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
    a. Use delimiters to clearly indicate distinct parts of the input
    b. Ask the model to check whether conditions are satisfied
    c. Give successful examples of completing tasks. Then ask the model to perform the task.
2. Give the model time to "think"
    a. Specify the steps required to complete a task
    b. 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.