Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

LAB Work 1

Course Code: COMP 314

Submitted By

Name: Amar Kumar Mandal

Roll No: 58

Level: 3rd year/ 2nd Semester

Group: CE

Submitted To

Name: Rajani Chulyadyo, PhD

DoCSE

Department of Computer Science and Engineering

Submission Date: March 6, 2022

1. Implementation of linear and binary search algorithm

Linear Search Algorithm

```
def linear_search(arr, x):
    for i in range(0, len(arr)):
        if(arr[i] == x):
            return i
    return -1
```

- Linear_search is the function that take a list of elements “arr” and target element “x” to be search in the list “arr”
- If the target element “x” is found in the list it returns the index of the target element otherwise –1

Binary Search Algorithm

```
def binary_search(arr, l, r, x):
    if r >= l:
        mid = l + (r - l) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, l, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, r, x)
    else:
        return -1
```

- binary_search is the function which take list of elements “arr”, lower index “l”, higher index “r” and target element “x” as a arguments
- If the target element is found in the list the function binary_search returns the index of the found element else -1.

2. Write some test cases to test your program

→ For each searching algorithm i.e linear search and binary search, 3 different test cases are written.

```
import unittest
from binary_search import binary_search
from linear_search import linear_search

class TestCase(unittest.TestCase):


   def test_linear_search(self):
       values = [1, 2, 3, 4, 5]
       not_equal_flag = -1
       self.assertNotEqual(linear_search(values, 2), not_equal_flag)
       self.assertNotEqual(linear_search(values, 3), not_equal_flag)
       self.assertNotEqual(linear_search(values, 99), not_equal_flag)


   def test_binary_search(self):
       values = [1, 2, 3, 4, 5]
       not_equal_flag = -1
       self.assertNotEqual(binary_search(values, 0, len(values) - 1, 1), not_equal_flag)
       self.assertNotEqual(binary_search(values, 0, len(values) - 1, 3), not_equal_flag)
       self.assertNotEqual(binary_search(values, 0, len(values) - 1, 99), not_equal_flag)


if __name__ == "__main__":
   unittest.main()
```

→When each test case is executed, the following outputs are generated.

Output

```
======================================================================
FAIL: test_binary_search (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_search.py", line 25, in test_binary_search
    self.assertNotEqual(binary_search(values, 0, len(values) - 1, 99), not_equal_fl
AssertionError: -1 == -1

======================================================================
FAIL: test_linear_search (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_search.py", line 17, in test_linear_search
    self.assertNotEqual(linear_search(values, 99), not_equal_flag)
AssertionError: -1 == -1

----------------------------------------------------------------------
Ran 2 tests in 0.000s

FAILED (failures=2)
(dsa-lab) amar@white-shadow:$ 
```

→ As in the above output, we can see that a total of 2 tests have been executed and there is one AssertionError in each test case since value **99** is not present in the list.

3. Generate some random inputs for your program and apply both linear and binary search algorithms to find a particular element on the generated input. Record the execution times of both algorithms for best and worst cases on inputs of different sizes (e.g. from 10000 to 100000 with step size as 10000). Plot an input-size vs execution-time graph.

→ Here the input size is taken in the range of 100 to 100000 with the step size of 10. After that the graph

plot.py > case_linear_search

```python
import matplotlib.pyplot as plt
import time
from binary_search import binary_search
from linear_search import linear_search


input_sizes = range(100, 100000, 10)
linear_best_case_time = []
linear_worst_case_time = []

binary_best_case_time = []
binary_worst_case_time = []

def case_binary_search(no_of_inputs, target):
    start_time = time.time()
    binary_search(range(no_of_inputs), 0, no_of_inputs - 1, target)
    end_time = time.time()
    diff = ( end_time - start_time) * 1000
    return diff

def case_linear_search(no_of_inputs, target):
    start_time = time.time()
    linear_search(range(no_of_inputs), target)
    end_time = time.time()
    diff = ( end_time - start_time) * 1000
    return diff
```

```python

# for linear search
for i in input_sizes:
    best_time_in_ms = case_linear_search(i, 0)
    worst_time_in_ms = case_linear_search(i, i)
    linear_best_case_time.append(best_time_in_ms)
    linear_worst_case_time.append(worst_time_in_ms)

# for binary search
for j in input_sizes:
    best_time_in_ms = case_binary_search(j, (j - 1) // 2)
    worst_time_in_ms = case_binary_search(j, j)
    binary_best_case_time.append(best_time_in_ms)
    binary_worst_case_time.append(worst_time_in_ms)

fig, (plt1, plt2) = plt.subplots(nrows=1, ncols=2)

plt1.plot(input_sizes, linear_best_case_time, ".", label="Best Case")
plt1.plot(input_sizes, linear_worst_case_time, "*", label="Worst Case")
plt1.set_xlabel("Input size")
plt1.set_ylabel("Time in miliseconds")
plt1.set_title("Linear Search")
plt1.legend()

```

```python

#binary search
plt2.plot(input_sizes, binary_best_case_time, ".", label="Best Case")
plt2.plot(input_sizes, binary_worst_case_time, "*", label="Worst Case")
plt2.set_xlabel("Input size")
plt2.set_ylabel("Time in miliseconds")
plt2.set_title("Binary Search")
plt2.legend()
plt.show()

def worst_case_linear_search(worst_case_value):
    start_time = time.time()
    linear_search(range(100), worst_case_value)
    end_time = time.time()
    diff = (start_time - end_time) * 1000

print(case_linear_search(1000))
```

Output:

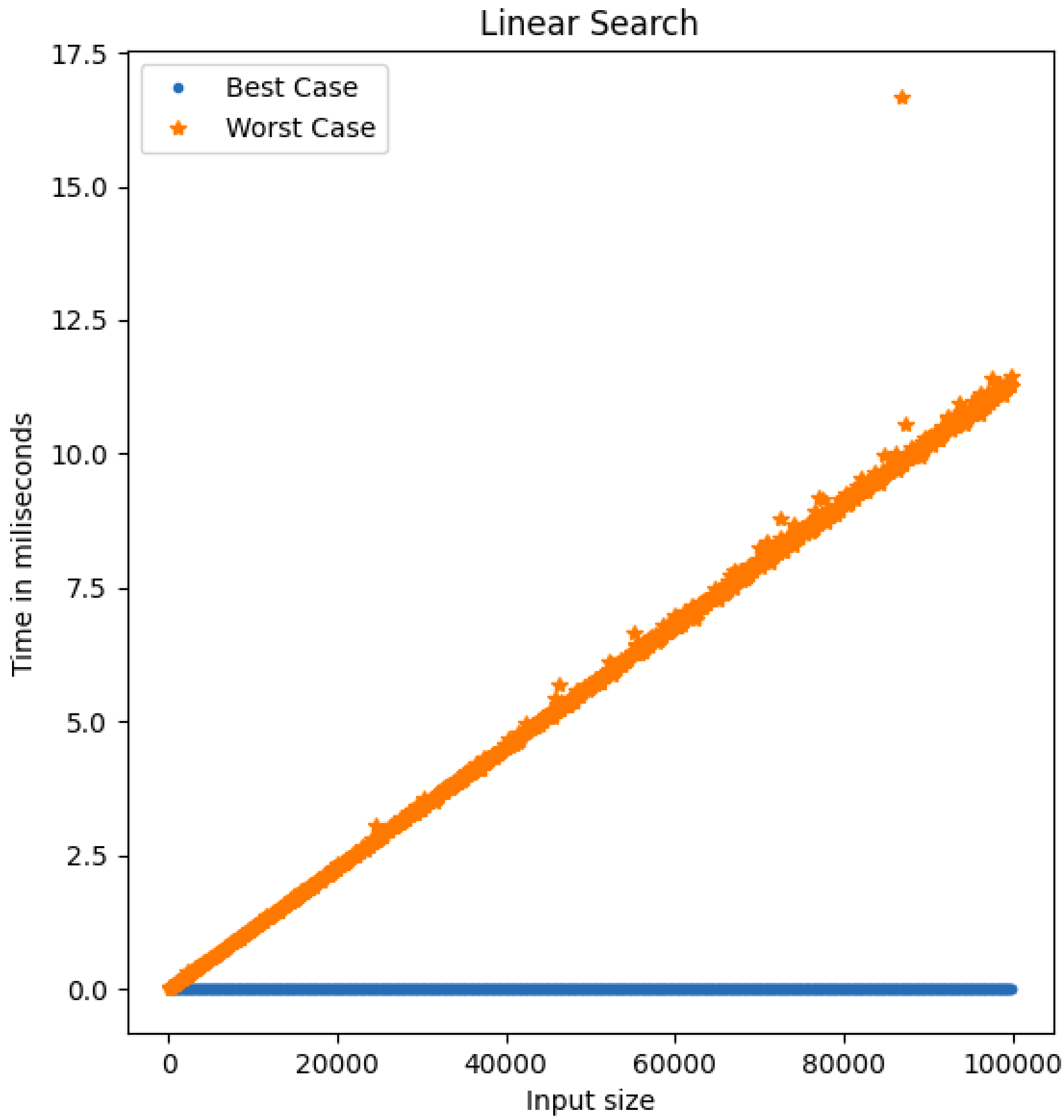

Fig 1: Input size vs computation time - Linear search

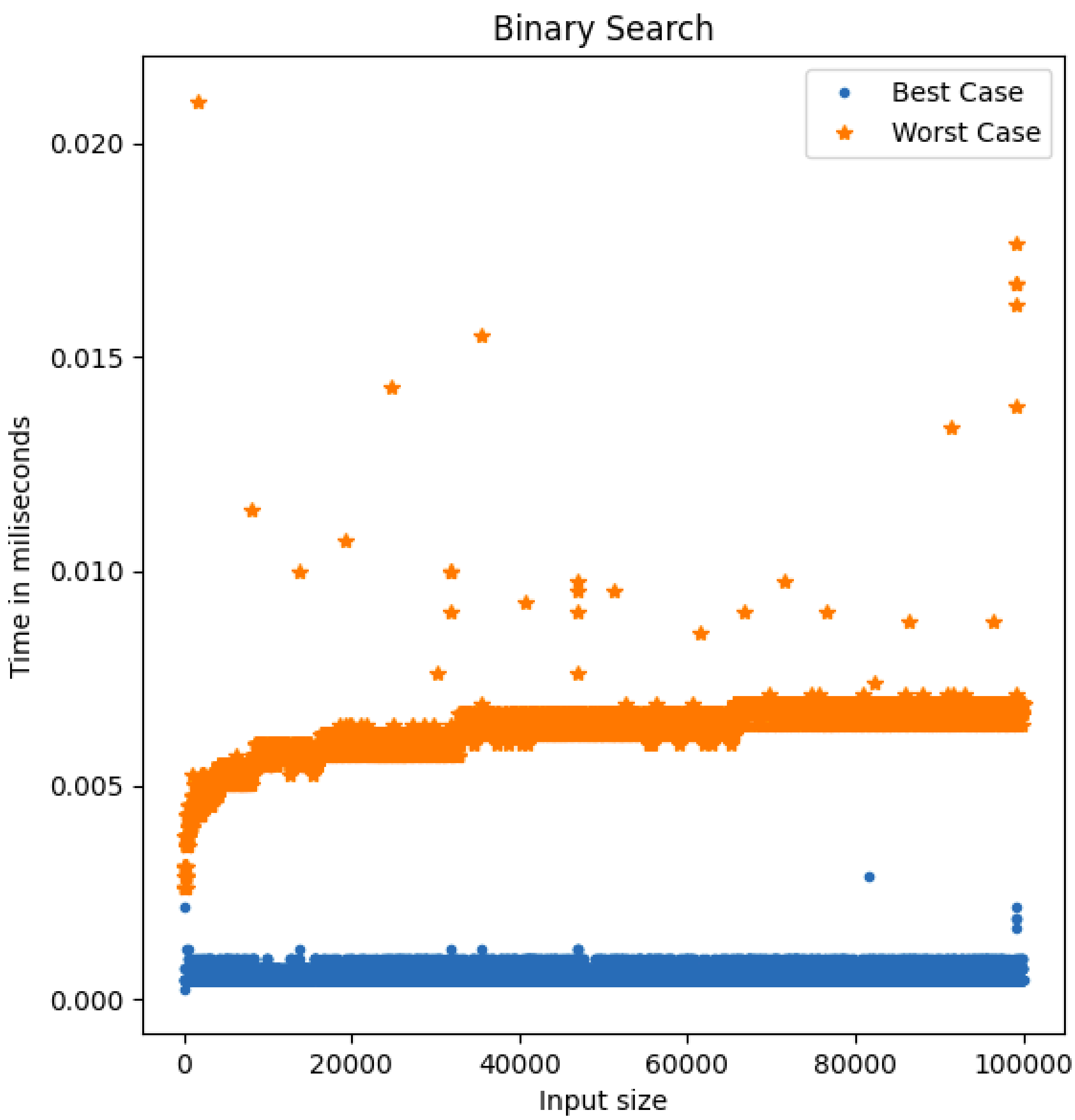

Fig 2: Input size vs computation time - Binary search

4. Explain your observations

→ Figure 1 shows the time complexity of the linear search algorithm when input size increases. As we know that the time complexity of the linear search algorithm in the best case scenario is **O(1)**, we can see in the graph that even though the size of the input increases the running time remains the same as indicated by the blue horizontal line. Similarly, for the worst case linear search algorithm is **O(n),** that is why as the size of input increases the running time increases proportionally to the input.

→ Figure 2 shows the best case and worst case time complexity of the binary search algorithm as the input size increases. The worst case and best case time complexity of binary search algorithms is **O(logn)** and **O(n)** respectively. So, every time input size doubles the running time increases by 1 unit in the worst case scenario and for best case it remains constant. When plotted in the graph we can indeed find that in the worst case the curve generated is of logarithmic nature whereas in the best case the curve is parallel to the x-axis which indicates the constant running time.