Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Lab Report 1

COMP 314

(For partial fulfillment of 3rd Year/ 2nd Semester in Computer Engineering)

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submitted by:

Neha Malla

C.E.

Roll No. 27

1. Implement linear and binary search algorithms.

➔ Filename: search.py

```
def linear_search(data, target):
    for i in range(len(data)):
        if data[i] == target:
            return i
    return -1


def binary_search(data, target):
    first = 0
    last = len(data) - 1

    while first <= last:
        mid = (first + last)//2
        if data[mid] == target:
            return mid
        if data[mid] < target:
            first =  mid + 1
        else:
            last = mid - 1
    return -1
```

2. Write some test cases to test your program.

➔ Filename: searchTest.py

```
import unittest

from search import linear_search

from search import binary_search


class TestSearch(unittest.TestCase):

    def test_search(self):

        data = [1, 2, 3, 5, 6, 12, 7, 4, 8]

        self.assertEqual(linear_search(data, 6), 4)

        self.assertEqual(linear_search(data, 10),  -1)

        self.assertEqual(binary_search(data, 6), 4)

        self.assertEqual(binary_search(data, 10),  -1)


    def test_searchChar(self):

        data = ['t', 'a', 'b', 'l', 'e']

        self.assertEqual(linear_search(data, 'a'), 1)

        self.assertEqual(binary_search(data, 'a'), 1)


if __name__ == "__main__":

    unittest.main()
```

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 size (e.g. from 10000 to 100000 with step size as 10000). Plot an input-size vs execution-time graph.

➔ Filename: searchMain.py

```
from search import linear_search

from search import binary_search

from time import time

import random

import matplotlib.pyplot as plt


def partition(arr, low, high):

    i = (low - 1)

    pivot = arr[high]

    for j in range(low , high):

        if arr[j] <= pivot:

            i = i + 1

            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]

    return i+1


def quickSort(arr, low, high):

    if low < high:

        q = partition(arr, low, high)

        quickSort(arr, low, q - 1)

        quickSort(arr, q + 1, high)
```

```
linearBest = []
linearWorst = []
binaryBest = []
binaryWorst = []

i = 10000
dataSize = []

while i <= 100000:
    dataSize.append(i)
    data = random.sample(range(i), i)

    start = time()
    indexLB = linear_search(data, data[0])
    end = time()
    linearBest.append(end - start)

    start = time()
    indexLW = linear_search(data, data[-1])
    end = time()
    linearWorst.append(end - start)

    quickSort(data, 0, len(data) - 1)

    start = time()
```

```
    indexBB = binary_search(data, data[(len(data) - 1)//2])

    end = time()

    binaryBest.append(end - start)


    start = time()

    indexBW = binary_search(data, data[-1])

    end = time()

    binaryWorst.append(end - start)


    i += 10000


print(indexLB)

print(indexLW)

print(indexBB)

print(indexBW)


print(linearBest)

print(linearWorst)

print(binaryBest)

print(binaryWorst)


plt.plot(dataSize,linearBest,"g")

plt.show()


plt.plot(dataSize,linearWorst,"r")

plt.show()
```

```
plt.plot(dataSize,binaryBest,"g")

plt.show()


plt.plot(dataSize,binaryWorst,"r")

plt.show()
```

4. Explain your observations.

- ➔ **Output:**

  Indexes for searched data value:

  Linear Best Case: 0

  Linear Worst Case: 99999

  Binary Best Case: 49999

  Binary Worst Case: 99999

  Time taken:

  Linear Best Case: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

  Linear Worst Case: [0.0, 0.0009973049163818359, 0.0019524097442626953, 0.001995086669921875, 0.003900766372680664, 0.003239154815673828, 0.003956317901611328, 0.005024433135986328, 0.0059163570404052734, 0.006980180740356445]

  Binary Best Case: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

  Binary Worst Case: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

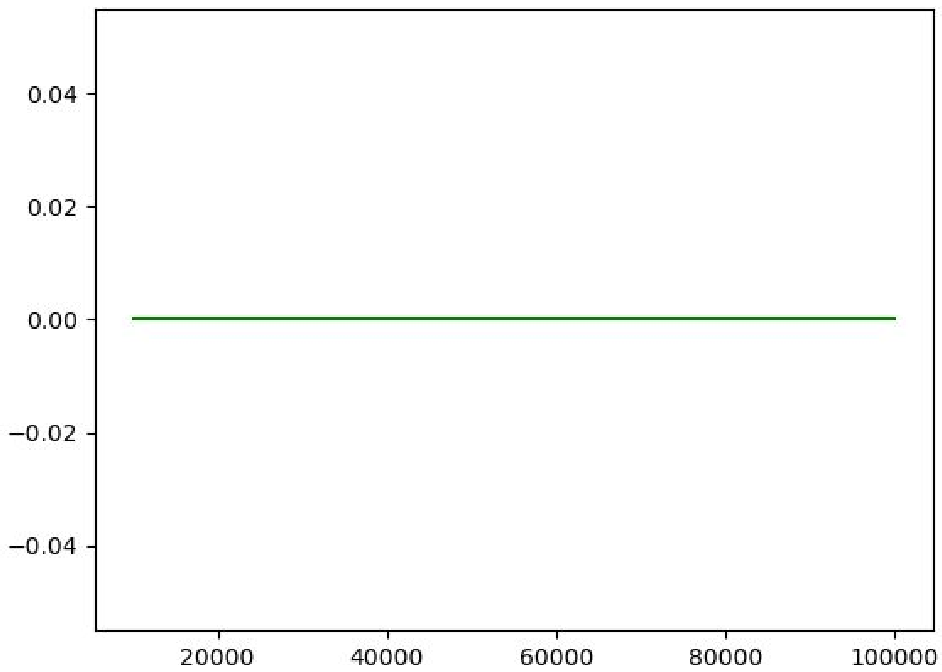

Figure: Linear Search Best Case Time Complexity

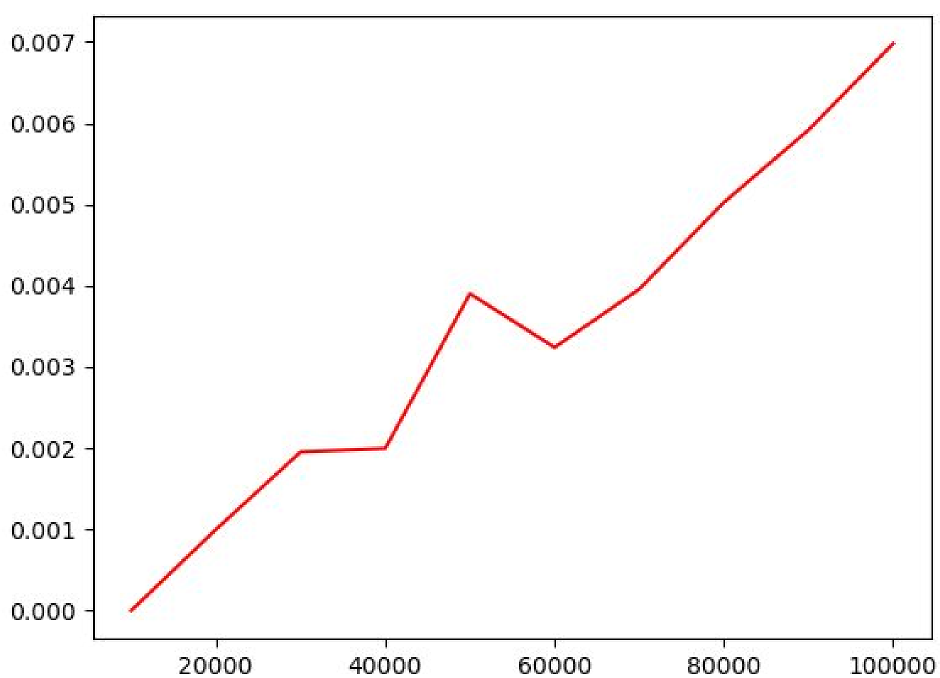

Figure: Linear Search Worst Case Time Complexity

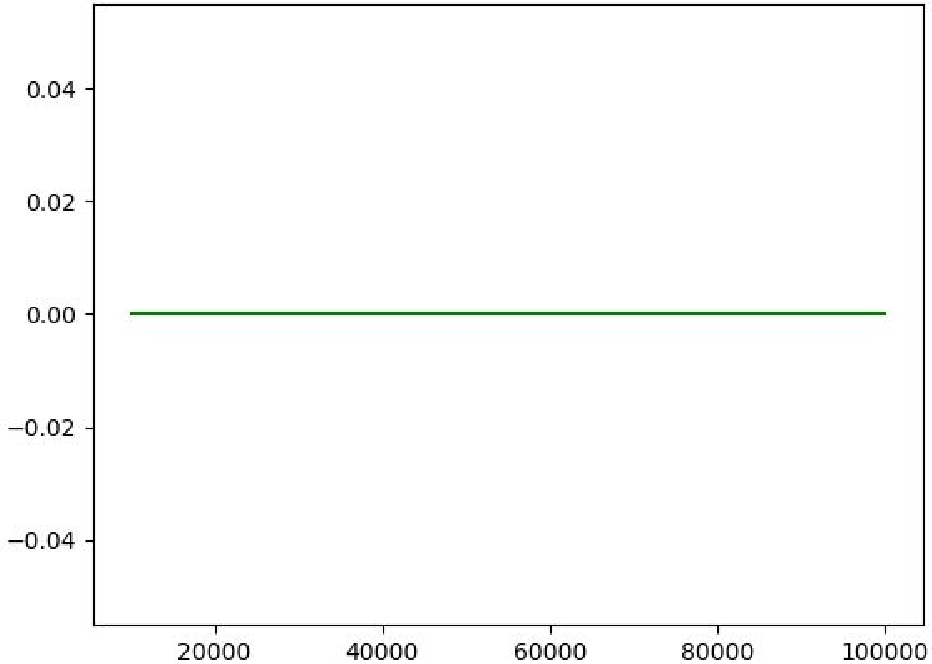

Figure: Binary Search Best Case Time Complexity

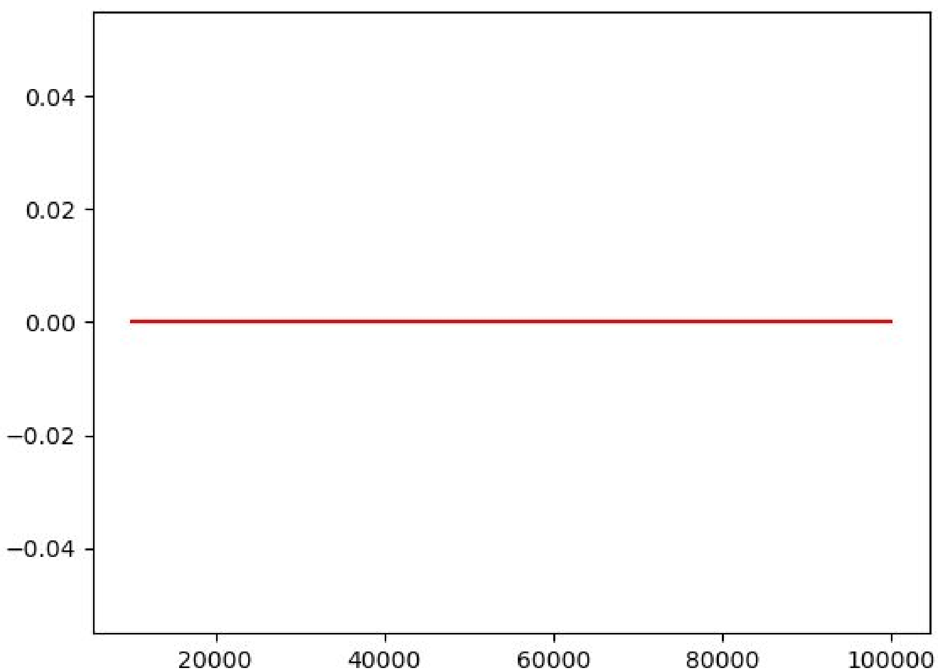

Figure: Binary Search Worst Case Time Complexity

Linear Search Best Case Time Complexity:

For this, the first element of the list is taken and searched. So the time complexity is O(1). Hence, a graph is a straight line parallel to x-axis as time is constant even when data size is increased.

Linear Search Worst Case Time Complexity:

For this, the last element of the list is taken and searched. So the time complexity is O(n). Hence, a graph is linear as time increases when data size is increased.

Binary Search Best Case Time Complexity:

For this, the middle element of the list is taken, as it is the root node in the binary tree and searched. So the time complexity is O(1). Hence, a graph is a straight line parallel to x-axis as time is constant even when data size is increased.

Binary Search Worst Case Time Complexity:

For this, the last element of the list is taken and searched. So the time complexity is $O(\log_2 n)$. Since the data size is small, a graph with constant time complexity is obtained, even when data size is increased.