KATHMANDU UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

LAB 1

A **Third year/ Second Semester** Algorithm and Complexity [COMP 314]
Lab 1 submitted in partial fulfillment of the requirements
for the degree of Bachelor of Engineering.

Submitted by:
Ashish Pokhrel
Faculty: C.E.
Roll: 38
Registration No: 022446-17

Submitted to:
Dr. Rajani Chulaydyo
Algorithm and Complexity (COMP 314)
Department of Computer Science and Engineering

Aug 5, 2020

Objectives:

- Implement Linear and Binary Search Algorithms.
- Write some test cases to test your program.
- Generate some random inputs for your program and apply both linear and binary search algorithms to and 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.

Introduction:

1. **Linear Search:**
   Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm:

```
Linear Search (Array A, Values x)

Step 1: Set counter to 1
Step 2: if counter > n then go to step 7
Step 3: if A[counter] = x then go to step 6
Step 4: Set counter to counter + 1
Step 5: Go to Step 2-Loop
Step 6: Print Element x Found at index counter and go to step 8
Step 7: Print element not found
Step 8: Exit
```

Pseudo Code:

```
procedure linear_search (list, value)
   for each item in the list
               if match item == value
                           return the item's location
                     end if
   end for
end procedure
```

Time Complexity:

Worst case time complexity: **O(N)**

Average case time complexity: **O(N)**
Best case time complexity: **O(1)**

2. Binary Search:

Binary search is a fast search algorithm with run-time complexity of O (log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Algorithm:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return -1

Pseudo Code:

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Time Complexity:

Worst case time complexity: **O(logN)**
Average case time complexity: **O(logN)**
Best case time complexity: **O(1)**

Observation:

At first, code for both linear and binary search was written. This was then tested using python unittest library. Tests ran successfully and the algorithm was validated. Then I move onwards to test the performance of both algorithms.
For linear search, 10000 to 300000 data points with an increment of 10000 each step were randomly generated in random order and simulated three cases: Best Case, Average Case and Worst-Case. For Best Case, we gave the algorithm to search for the first element in the list. For the average case, we randomly generated the data to be searched. For the worst case, we gave it the last data on the list.

For binary search, 10000 to 10000000 data points were used as this algorithm is significantly faster than previous algorithm and time measurements were done in nano seconds for the same reason. Since for binary search requires data to be sorted in ascending order, we generated numbers in ascending order. Similar to previous method, three tests were conducted. For Best Case, list's middle element was given as search value. For average case, we generated random value to be searched. For worst case, we generated data as complete binary tree and fed the last leaf node as search value.

Output:

Linear Search (In Seconds):

Worst Case
Elapsed Time for 10000 datas: 0.000798
Elapsed Time for 20000 datas: 0.001988

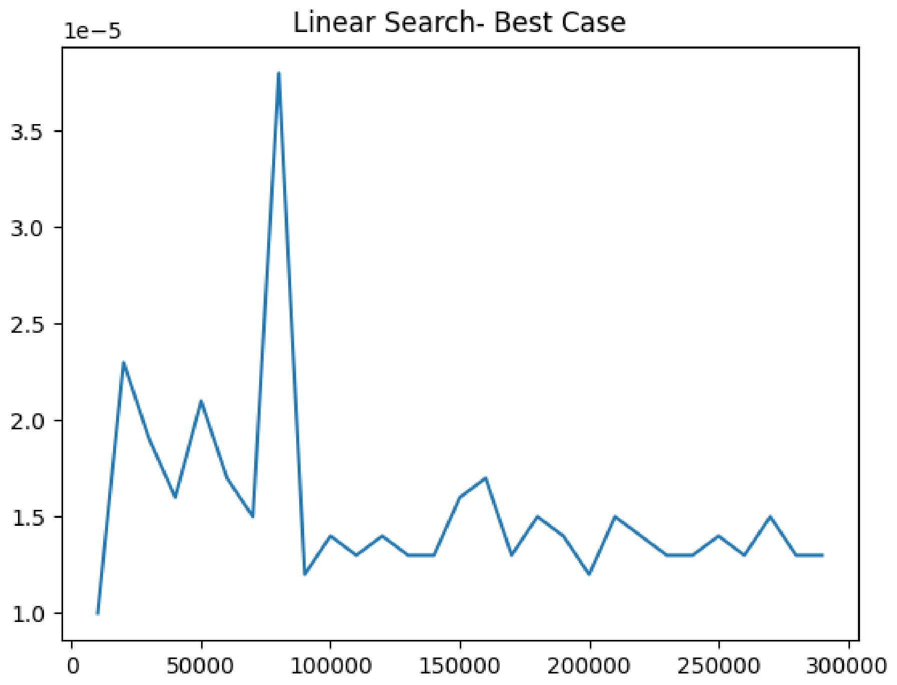
```
Elapsed Time for 30000 datas: 0.002228
Elapsed Time for 40000 datas: 0.003177
Elapsed Time for 50000 datas: 0.00331
Elapsed Time for 60000 datas: 0.003926
Elapsed Time for 70000 datas: 0.004514
Elapsed Time for 80000 datas: 0.007507
Elapsed Time for 90000 datas: 0.013666
Elapsed Time for 100000 datas: 0.013993
Elapsed Time for 110000 datas: 0.056185
Elapsed Time for 120000 datas: 0.017898
Elapsed Time for 130000 datas: 0.02367
Elapsed Time for 140000 datas: 0.055672
Elapsed Time for 150000 datas: 0.028391
Elapsed Time for 160000 datas: 0.032385
Elapsed Time for 170000 datas: 0.036983
Elapsed Time for 180000 datas: 0.029536
Elapsed Time for 190000 datas: 0.026411
Elapsed Time for 200000 datas: 0.037
Elapsed Time for 210000 datas: 0.035892
Elapsed Time for 220000 datas: 0.0309
Elapsed Time for 230000 datas: 0.032893
Elapsed Time for 240000 datas: 0.059181
Elapsed Time for 250000 datas: 0.060668
Elapsed Time for 260000 datas: 0.066853
Elapsed Time for 270000 datas: 0.03909
Elapsed Time for 280000 datas: 0.040097
Elapsed Time for 290000 datas: 0.050133
```

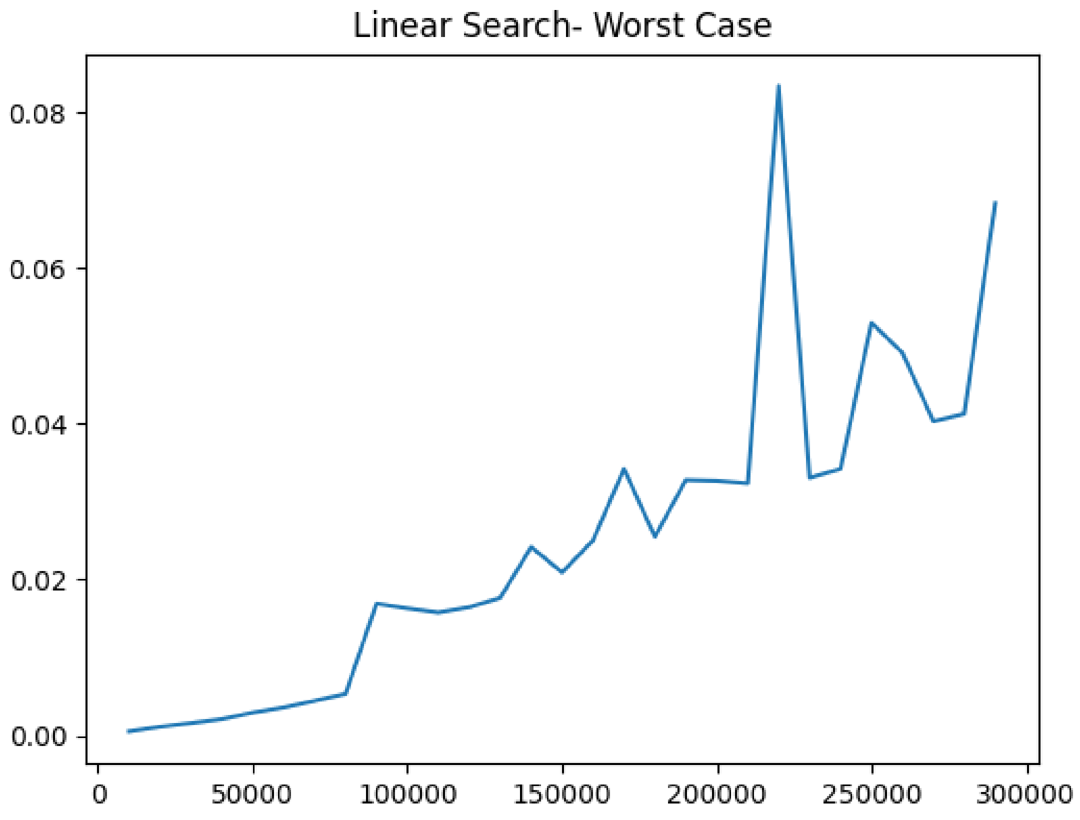
Best Case

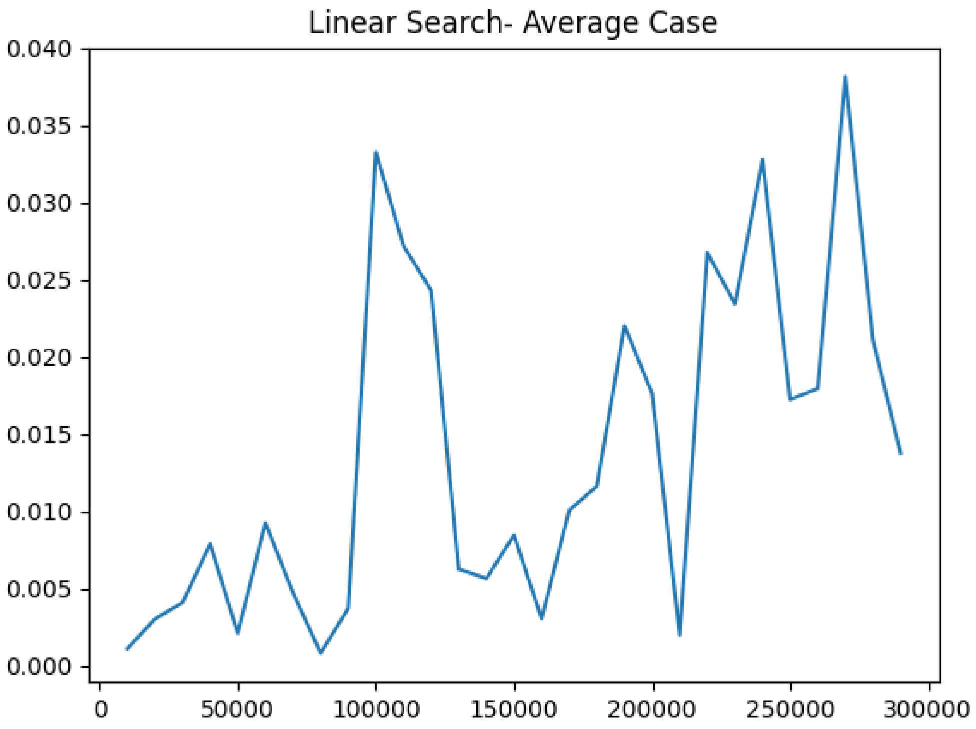
```
Elapsed Time for 10000 datas: 1e-05
Elapsed Time for 20000 datas: 2.3e-05
Elapsed Time for 30000 datas: 1.9e-05
Elapsed Time for 40000 datas: 1.6e-05
Elapsed Time for 50000 datas: 2.1e-05
Elapsed Time for 60000 datas: 1.7e-05
Elapsed Time for 70000 datas: 1.5e-05
Elapsed Time for 80000 datas: 3.8e-05
Elapsed Time for 90000 datas: 1.2e-05
Elapsed Time for 100000 datas: 1.4e-05
Elapsed Time for 110000 datas: 1.3e-05
Elapsed Time for 120000 datas: 1.4e-05
Elapsed Time for 130000 datas: 1.3e-05
Elapsed Time for 140000 datas: 1.3e-05
Elapsed Time for 150000 datas: 1.6e-05
Elapsed Time for 160000 datas: 1.7e-05
Elapsed Time for 170000 datas: 1.3e-05
Elapsed Time for 180000 datas: 1.5e-05
Elapsed Time for 190000 datas: 1.4e-05
Elapsed Time for 200000 datas: 1.2e-05
Elapsed Time for 210000 datas: 1.5e-05
Elapsed Time for 220000 datas: 1.4e-05
Elapsed Time for 230000 datas: 1.3e-05
Elapsed Time for 240000 datas: 1.3e-05
Elapsed Time for 250000 datas: 1.4e-05
Elapsed Time for 260000 datas: 1.3e-05
Elapsed Time for 270000 datas: 1.5e-05
Elapsed Time for 280000 datas: 1.3e-05
Elapsed Time for 290000 datas: 1.3e-05
```

Average Case

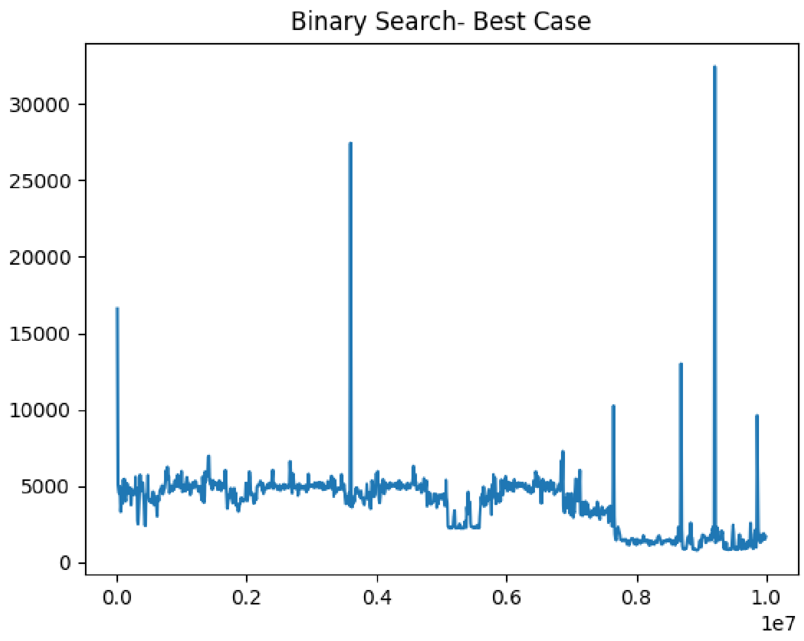
```
Elapsed Time for 10000 datas: 0.001105
Elapsed Time for 20000 datas: 0.003032
Elapsed Time for 30000 datas: 0.004094
Elapsed Time for 40000 datas: 0.007881
Elapsed Time for 50000 datas: 0.002101
Elapsed Time for 60000 datas: 0.009247
Elapsed Time for 70000 datas: 0.004724
Elapsed Time for 80000 datas: 0.000818
Elapsed Time for 90000 datas: 0.003721
Elapsed Time for 100000 datas: 0.033262
Elapsed Time for 110000 datas: 0.027192
Elapsed Time for 120000 datas: 0.024303
Elapsed Time for 130000 datas: 0.006267
Elapsed Time for 140000 datas: 0.005651
Elapsed Time for 150000 datas: 0.00846
Elapsed Time for 160000 datas: 0.003054
Elapsed Time for 170000 datas: 0.010056
Elapsed Time for 180000 datas: 0.011632
Elapsed Time for 190000 datas: 0.022023
Elapsed Time for 200000 datas: 0.017633
Elapsed Time for 210000 datas: 0.001985
Elapsed Time for 220000 datas: 0.026751
Elapsed Time for 230000 datas: 0.023426
Elapsed Time for 240000 datas: 0.03277
Elapsed Time for 250000 datas: 0.017235
Elapsed Time for 260000 datas: 0.017954
Elapsed Time for 270000 datas: 0.038151
Elapsed Time for 280000 datas: 0.021099
Elapsed Time for 290000 datas: 0.013775
```

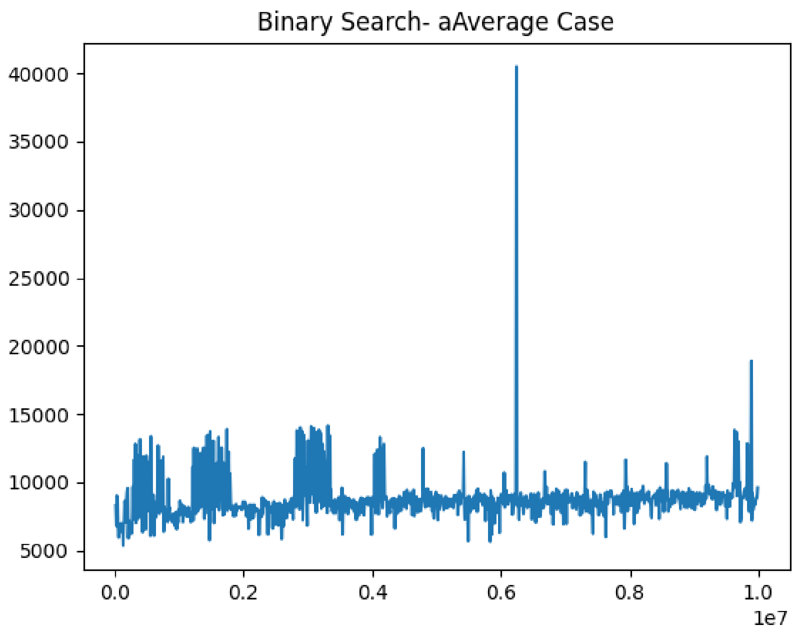

Linear Search- Best Case
1e−5
3.5
3.0
2.5
2.0
1.5
1.0
0
50000
100000
150000
200000
250000
300000

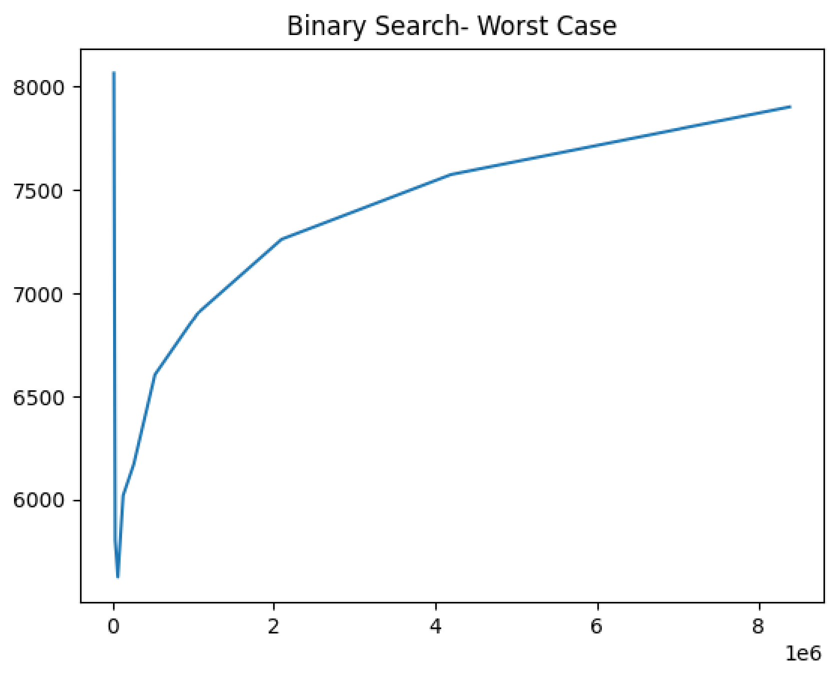

Linear Search- Worst Case
0.08
0.06
0.04
0.02
0.00
0
50000
100000
150000
200000
250000
300000


Linear Search- Average Case
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
50000
100000
150000
200000
250000
300000

Binary Search (In ns):
As there are too many time points, we'll only show plots of the result.







Conclusion:

From the above time measurements as well as graphs we can see that, in linear search, for worst case the results grows linearly as number of points increases. So the time complexity can be interpreted as O(n). For binary search, it can be seen that time increases logarithmitically. So the time complexity is O(log(n)).

Source Code:

Search.py

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


def binary_search(data, value):
    l, r = 0, len(data)-1

    while(l <= r):
        mid = (l+r)//2
        if(data[mid] == value):
            return mid
        elif value > data[mid]:
            l = mid+1
        elif value < data[mid]:
            r = mid-1
    return -1
```

SearchMain.py

```python
from search import linear_search, binary_search
from time import time, time_ns
import matplotlib.pyplot as plt
import math

import random
import numpy as np


class LinearSearch:
    def __init__(self):
        self.plottingDatas = []
        self.MAX = 300000
        self.MIN = 10000
        self.STEP = 5000


    def clearData(self):
        self.plottingDatas = []
    def plotData(self, title,color):
        if(len(self.plottingDatas) <= 0):
            raise Exception("No data found")
        plt.plot(*zip(*self.plottingDatas),c=color)
        plt.title(title)
        plt.show()

    def testBestCase(self):
        print("Best Case")
        self.clearData()
        for length in range(self.MIN, self.MAX, self.STEP):
            data = random.sample(range(self.MAX), length)
            start = time()
            linear_search(data, data[0])
            end = time()
            elapsed = end-start
            print(f"Elapsed Time for {length} datas: {round(elapsed,6)}")
            self.plottingDatas.append((length, round(elapsed,6)))
        # print("Average Best Case: ",np.mean( list(zip(*self.plottingDatas))[1])
)
        self.plotData("Linear Search- Best Case",'green')

    def testWorstCase(self):
        print("Worst Case")
        self.clearData()
        for length in range(self.MIN, self.MAX, self.STEP):
            data = random.sample(range(self.MAX), length)
            start = time()
            linear_search(data, data[-1])
            end = time()
            elapsed = end-start
            print(f"Elapsed Time for {length} datas: {round(elapsed,6)}")
            self.plottingDatas.append((length, round(elapsed,6)))

        # print("Average Worst Case: ",np.mean( list(zip(*self.plottingDatas))[1]
))

        self.plotData("Linear Search- Worst Case",'blue')

    def testAverageCase(self):
        print("Average Case")
        self.clearData()
        for length in range(self.MIN, self.MAX, self.STEP):
            data = random.sample(range(self.MAX), length)
            r = random.randint(0, length)
            start = time()
            linear_search(data, data[r])
            end = time()
            elapsed = end-start
            print(f"Elapsed Time for {length} datas: {round(elapsed,6)}")
            self.plottingDatas.append((length, round(elapsed,6)))

        # print("Average Average Case: ",np.mean( list(zip(*self.plottingDatas))[
1]))

        self.plotData("Linear Search- Average Case",'red')


class BinarySearch:
    def __init__(self):
        self.plottingDatas = []
        self.MAX = 10000000
        self.MIN = 10000
        self.STEP = 10000

    def clearData(self):
        self.plottingDatas = []

    def plotData(self, title):
        if(len(self.plottingDatas) <= 0):
            raise Exception("No data found")
        plt.plot(*zip(*self.plottingDatas))
        plt.title(title)
        plt.show()

    def testBestCase(self):
        print("Best Case")
        self.clearData()
        for length in range(self.MIN, self.MAX, self.STEP):
            data = range(length)
            start = time_ns()
            binary_search(data, data[length//2-1])
            end = time_ns()
            elapsed = end-start
            print(f"Elapsed Time for {length} datas: {round(elapsed,6)}")
            self.plottingDatas.append((length, round(elapsed,6)))

        # print("Average Best Case: ",np.mean( list(zip(*self.plottingDatas))[1])
)
```

```
        self.plotData("Binary Search- Best Case")

    def testWorstCase(self):
        print("Worst Case")
        self.clearData()
        k = int(math.log2(self.MIN))+1
        n = 2 ** k - 1

        while(n <= self.MAX):
            data = range(n)
            start = time_ns()
            binary_search(data, data[-1])
            end = time_ns()
            elapsed = end-start
            print(f"Elapsed Time for {n} datas: {round(elapsed,6)}")
            self.plottingDatas.append((n, round(elapsed,6)))

            k += 1
            n = 2 ** k - 1

        # print("Average Worst Case: ",np.mean( list(zip(*self.plottingDatas))[1]
))
        self.plotData("Binary Search- Worst Case")

    def testAverageCase(self):
        print("Average Case")
        self.clearData()
        for length in range(self.MIN, self.MAX, self.STEP):
            data = range(length)
            r = random.randint(0, length)
            start = time_ns()
            rand = data[r]
            # binary_search(data, data[r])
            binary_search(data, rand)
            end = time_ns()
            elapsed = end-start
            print(f"Elapsed Time for {length} datas: {round(elapsed,6)}")
            self.plottingDatas.append((length, round(elapsed,6)))

        # print("Average Average Case: ",np.mean( list(zip(*self.plottingDatas))[
1]))
        self.plotData("Binary Search- aAverage Case")
```

```python
if __name__ == "__main__":
    # LinearSearch().testAverageCase()
    # LinearSearch().testBestCase()
    LinearSearch().testWorstCase()
    # BinarySearch().testWorstCase()
    # BinarySearch().testBestCase()
    # BinarySearch().testAverageCase()
    # BinarySearch().testWorstCase()
```

searchTest.py

```python
import unittest

from search import linear_search, binary_search


class TestLinearSearchMethods(unittest.TestCase):
    def testLinear(self):
        data = [1, 2, 3, 4, 5]
        self.assertEqual(linear_search(data, 2), 1)
        self.assertEqual(linear_search(data, 10), -1)
    def testChar(self):
        data = ['a','e','i','o','u']
        self.assertEqual(linear_search(data, 'a'), 0)
        self.assertEqual(linear_search(data, 'z'), -1)
        self.assertEqual(linear_search(data, 'u'), 4)



class TestBinarySearch(unittest.TestCase):
    def testBinary(self):
        data = [1, 2, 3, 4, 5]
        self.assertEqual(binary_search(data, 2), 1)
        self.assertEqual(binary_search(data, 10), -1)

    def testChar(self):
        data = ['a','e','i','o','u']
        self.assertEqual(binary_search(data, 'a'), 0)
        self.assertEqual(binary_search(data, 'z'), -1)
        self.assertEqual(binary_search(data, 'u'), 4)

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

Source Code available at Github: https://github.com/meashishpokhrel/Algorithm-Complexity-Lab