

Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre



A Lab Report
On
“Searching”
[Course Code: COMP 314]

Submitted by:

Ankit Khatiwada (24)

Submitted to:

Dr. Rajani Chulyadyo
Department of Computer Science and Engineering

7th March, 2022

Table of Contents

Objective	1
Procedure	1
Test case for algorithms	2
Plotting the running time of algorithms vs number of data	4
Conclusion	5
Appendix	6

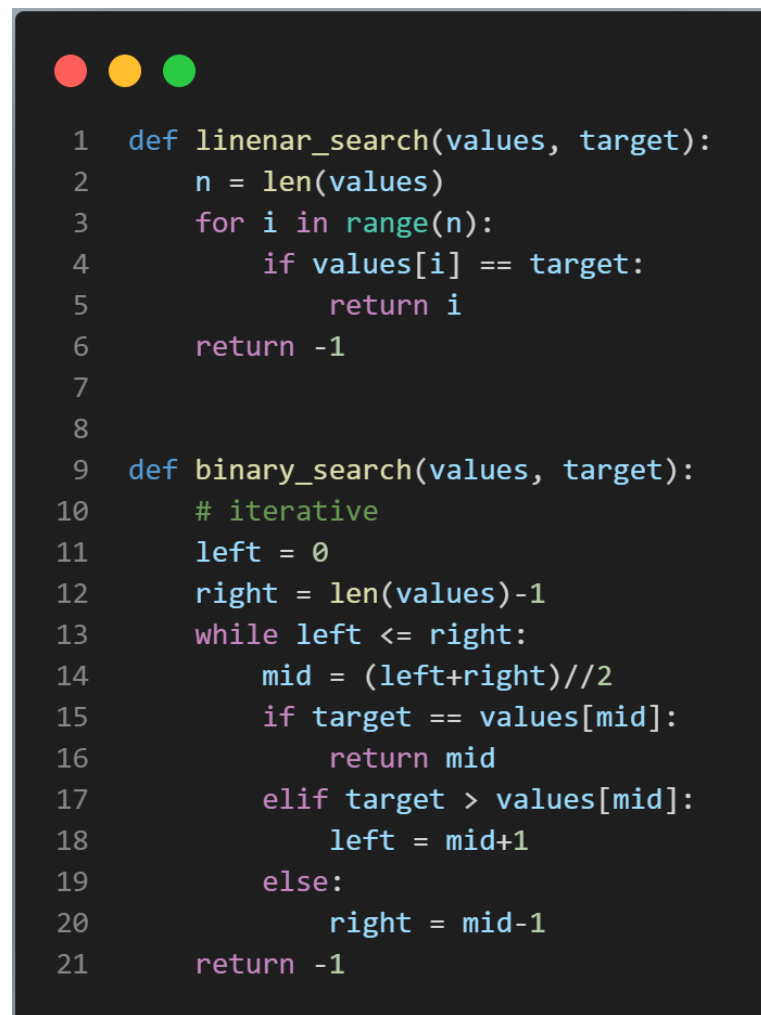
Objective

The task for the first lab was to write a program to search for data in the list. For this, we use two searching algorithms: linear and binary search and compare the time complexities of the two algorithms.

Procedure

At first, code for both linear and binary search was written. This was then tested using python's unit test library. Tests ran successfully and the algorithm was validated. Then moving towards to test the performance of both algorithms. For linear search, 100 to 100000 data points with an increment of 100 each step were generated and simulated for two cases: Best Case and Worst-Case. For Best Case, we gave the algorithm to search for the first element in the list. For the worst case, we gave it the data not on the list. For binary search, 100 to 1000000 data points were used as this algorithm is significantly faster than previous algorithm and time measurements were done in microseconds for the same reason. Since binary search requires data to be sorted in ascending order, we generated numbers in ascending order. Similar to the previous method, two tests were conducted. For Best Case, the list's middle element was given as a search value. For the worst case, we generated data as a complete binary tree and fed the last leaf node as search value.

The snapshot of the algorithms is as listed below:

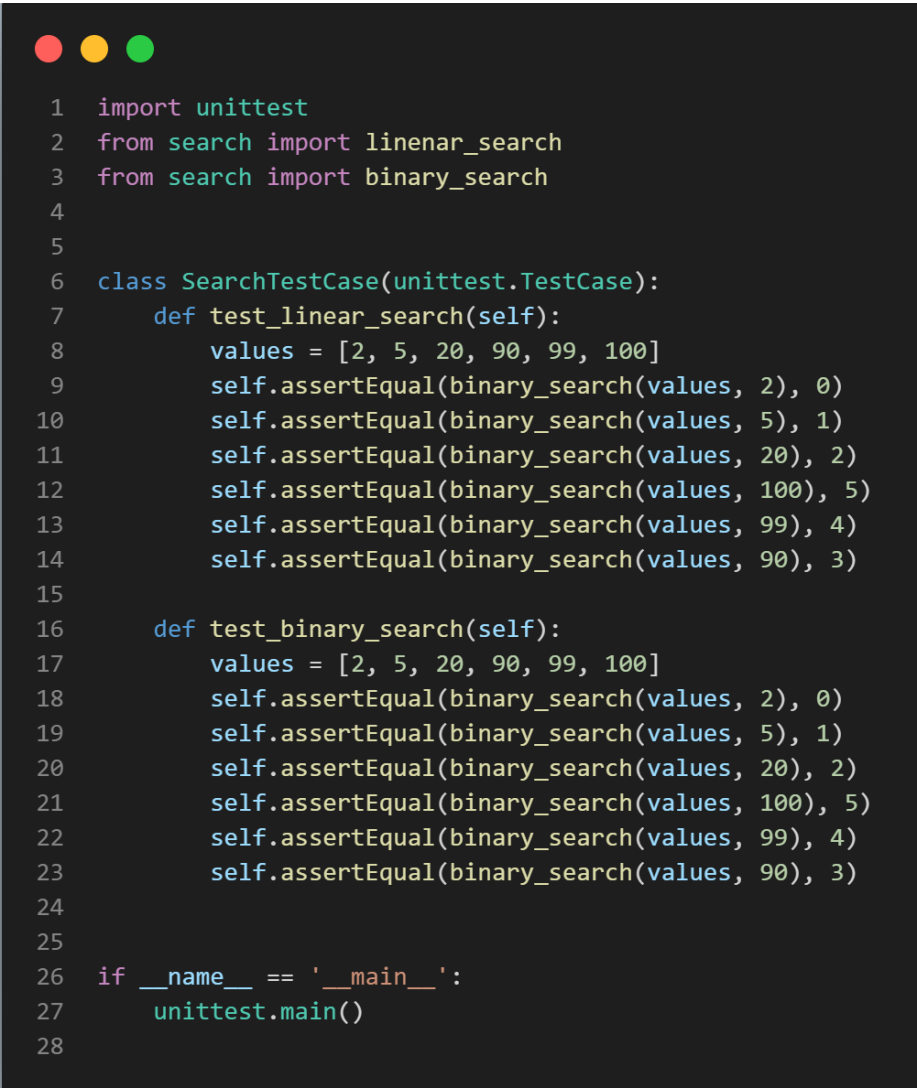


```
1 def linear_search(values, target):
2     n = len(values)
3     for i in range(n):
4         if values[i] == target:
5             return i
6     return -1
7
8
9 def binary_search(values, target):
10     # iterative
11     left = 0
12     right = len(values)-1
13     while left <= right:
14         mid = (left+right)//2
15         if target == values[mid]:
16             return mid
17         elif target > values[mid]:
18             left = mid+1
19         else:
20             right = mid-1
21     return -1
```

Fig 1: Algorithms for Linear and Binary Search

Test case for algorithms

The unittest standard library of the Python programming language was used for performing unit tests on the search algorithms as shown in the figure below. A random self generated list was taken and tests were performed for data present in or not in the list. Test methods for both the algorithms were passed and the test was validated:

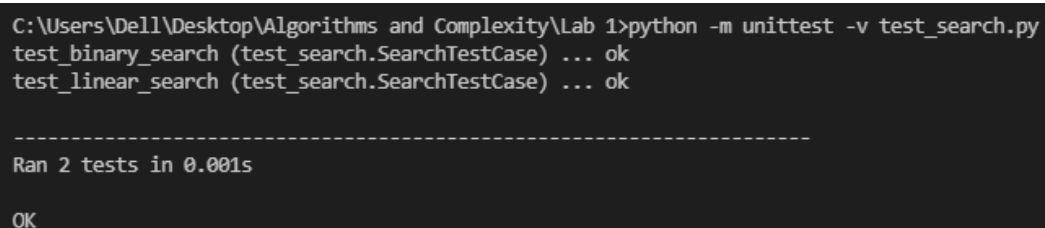


```

1  import unittest
2  from search import linear_search
3  from search import binary_search
4
5
6  class SearchTestCase(unittest.TestCase):
7      def test_linear_search(self):
8          values = [2, 5, 20, 90, 99, 100]
9          self.assertEqual(binary_search(values, 2), 0)
10         self.assertEqual(binary_search(values, 5), 1)
11         self.assertEqual(binary_search(values, 20), 2)
12         self.assertEqual(binary_search(values, 100), 5)
13         self.assertEqual(binary_search(values, 99), 4)
14         self.assertEqual(binary_search(values, 90), 3)
15
16         def test_binary_search(self):
17             values = [2, 5, 20, 90, 99, 100]
18             self.assertEqual(binary_search(values, 2), 0)
19             self.assertEqual(binary_search(values, 5), 1)
20             self.assertEqual(binary_search(values, 20), 2)
21             self.assertEqual(binary_search(values, 100), 5)
22             self.assertEqual(binary_search(values, 99), 4)
23             self.assertEqual(binary_search(values, 90), 3)
24
25
26  if __name__ == '__main__':
27      unittest.main()
28

```

Fig 2: Test cases for Linear search and Binary Search



```

C:\Users\Dell\Desktop\Algorithms and Complexity\Lab 1>python -m unittest -v test_search.py
test_binary_search (test_search.SearchTestCase) ... ok
test_linear_search (test_search.SearchTestCase) ... ok

-----
Ran 2 tests in 0.001s

OK

```

Fig 3: Result of unit test

Plotting the running time of algorithms vs number of data

Arrays with varying sizes were used to perform both the linear and binary search of an element in each array. Arrays of length ranging from 100 to 100000 were taken at an interval of 100 for linear search and 100 to 1000000 were taken at an interval of 100.

```
1 import matplotlib.pyplot as plt
2 import time
3
4 from search import linear_search
5 from search import binary_search
6
7 fig, (plt1, plt2) = plt.subplots(nrows=1, ncols=2)
8
9
10 def linear_search_complexity():
11     bestCaseList = []
12     worstCaseList = []
13     randomList = range(100, 100000, 100)
14
15     # For Linear Search best case
16     for num in randomList:
17         bestCaseStartTime = time.time()
18         linear_search(range(num), 0)
19         bestCaseEndTime = time.time()
20         bestCaseList.append((bestCaseEndTime-bestCaseStartTime)*1000*1000)
21     # For Linear Search Worst case
22     worstCaseStartTime = time.time()
23     linear_search(range(num), -5)
24     worstCaseEndTime = time.time()
25     worstCaseList.append((worstCaseEndTime-worstCaseStartTime)*1000*1000)
26
27     plt1.set_title("Linear Search")
28     plt1.set_xlabel("Array Size")
29     plt1.set_ylabel("Time(in microseconds)")
30     plt1.plot(randomList, bestCaseList, '*', label="Best Case")
31     plt1.plot(randomList, worstCaseList, '.', label="Worst Case")
32     plt1.legend()
33
34
35 def binary_search_complexity():
36     bestCaseList = []
37     worstCaseList = []
38     randomList = range(100, 1000000, 100)
39     # For Linear Search best case
40     for num in randomList:
41         bestCaseStartTime = time.time()
42         binary_search(range(num), (num-1)//2)
43         bestCaseEndTime = time.time()
44         bestCaseList.append((bestCaseEndTime-bestCaseStartTime)
45                             * 1000*1000) # Append time in best case time in microseconds
46     worstCaseStartTime = time.time()
47     binary_search(range(num), num)
48     worstCaseEndTime = time.time()
49     worstCaseList.append((worstCaseEndTime-worstCaseStartTime)
50                         * 1000*1000) # Append time in worst case time in microseconds
51
52     plt2.set_title("Binary Search")
53     plt2.set_xlabel("Array Size")
54     plt2.set_ylabel("Time(in microseconds)")
55     plt2.plot(randomList, bestCaseList, '.', label="Best Case")
56     plt2.plot(randomList, worstCaseList, '*', label="Worst Case")
57     plt2.legend()
58
59
60 if __name__ == '__main__':
61     linear_search_complexity()
62     binary_search_complexity()
63     plt.show()
```

Fig 4: Code for plotting the best and worst case time complexities of linear and binary search

The graph for time complexities was then plotted from the matplotlib package of Python as shown in the figure below



Fig 5: Running time graph of Linear and Binary Search for best and worst case

Conclusion

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

Appendix

Source Code: <https://github.com/ch-ankit/algo-and-complexity>