CSE 1384

Search Algorithm Timing

Lab #4

By

Cameron Gruich

CJG325

and

Ramesh Shrestha

RS2401

"On my honor, as a Mississippi State University student, I have neither given nor received unauthorized assistance on this academic work."

Student Signature _____

CSE 1384 Intermediate Programming

Class Section #2

Lab Instructor: Nieves-Rivera, Delma

September 11th, 2017

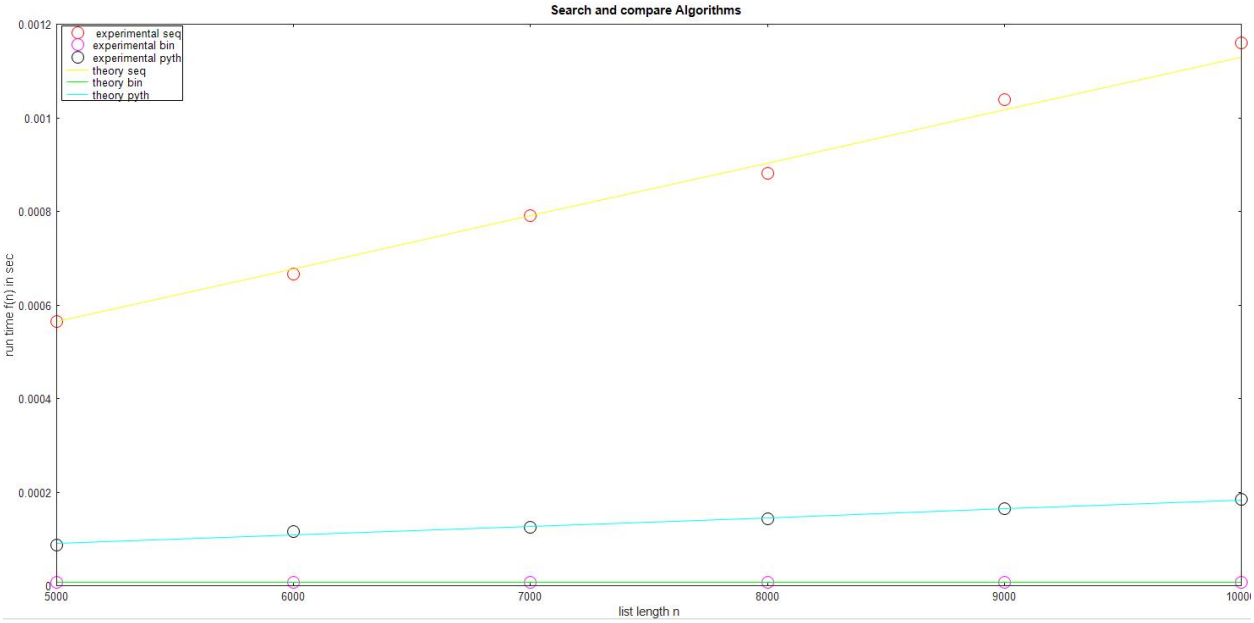
Timing

Timing Table

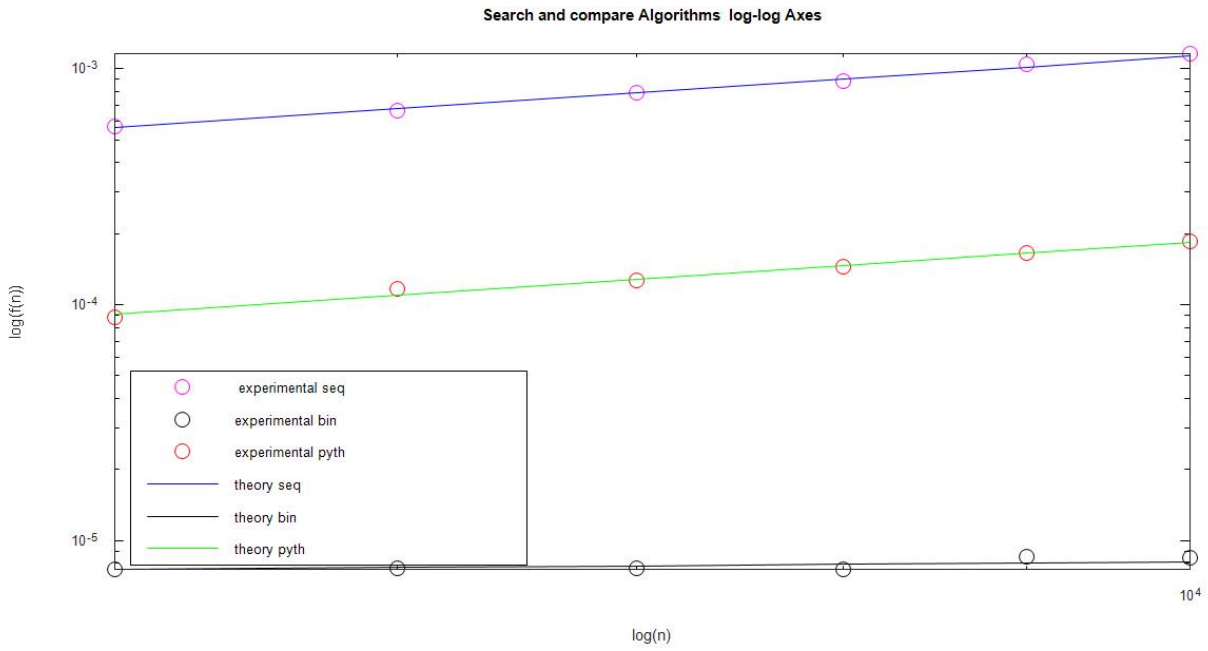
Sample Size (List Length n)	(Time In Seconds, C Value)		
	Sequential Search	Binary Search	Python Built-In Search
5000	(5.66e-4, 1.13e-7)	(7.51e-6, 6.11e-7)	(8.85e-5, 1.77e-8)
6000	(6.66e-4, 1.11e-7)	(7.61e-6, 6.06e-7)	(1.17e-4, 1.95e-8)
7000	(7.92e-4. 1.13e-7)	(7.61e-6, 5.96e-7)	(1.26e-4, 1.80e-8)
8000	(8.82e-4, 1.10e-7)	(7.51e-6, 5.79e-7)	(1.44e-4, 1.81e-8)
9000	(1.04e-3, 1.16e-7)	(8.51e-6, 6.48e-7)	(1.66e-4, 1.85e-8)
10000	(1.16e-3, 1.16e-7)	(8.41e-6, 6.33e-7)	(1.85e-4, 1.85e-8)
Avg C Value (Theoretical C)	1.13e-7	6.12e-7	1.83e-8

Timing Graph

Normal Plot



LogLog Plot



Conclusions

1) Which search algorithm was fastest?

The binary search algorithm was fastest due to logarithm base 2 run time. This run time was far faster than the linear run times.

2) Of these two, python search and your sequential search, which was the fastest?

Python's built-in search was faster than the sequential search function.

3) Does your timing data match your predictions? Back up your answer with tables and graphs.

Yes, the timing data matched the predictions. As is seen in the graphs, the experimental data fit the theoretical run-time curve well. Moreover, the C values for each algorithm were relatively constant in the data tables.

Code Appendix

Python Code

```
# Programmers: Cameron Gruich (cjg325) and Ramesh Shrestha (rs2401)

# import randint to generate random numbers for our list
from random import randint

# import time to record function run times
from time import time

# import log2
from math import log2

# Function: binary_search(listVar, numVar)
# Purpose: Performn a binary search on an ordered list for a desired
value.
# Paramters: listVar (The list to look through), numVar (The number to
look for)
#_____________________________________________________________________
_____________
def binary_search(listVar, numVar):

    listLength = len(listVar)

    l_index = 0
    r_index = listLength - 1

    while l_index <= r_index:
        mid_index = int((l_index + r_index)/2)

        if listVar[mid_index] == numVar:
            choice = mid_index

            # break the loop if the desired value has been found
            break

        elif listVar[mid_index] > numVar:
            # If the desired value is less than the middle index, stop
looking to the right.
            r_index = mid_index - 1

        else:
            # If the desired value is more than the middle index, stop
looking to the left.
            l_index = mid_index + 1

        if l_index > r_index:
            # return -1 for choice if the value is not found in the
list
            choice = -1
```

```
    return choice
#_____________________________________________________________________
_________________


# Function: sequential_search(listVar, valueVar)
# Purpose: Find a desired value in a list via sequential searching
# Parameters: listVar (the list we want to search in) and valueVar
# (the value we want to look for)
# _____________________________________________
def sequential_search(listVar, valueVar):

    listLen = len(listVar)

    for ind in range(0, listLen):

        if listVar[ind] == valueVar:
            choice = listVar.index(valueVar)

            # Break the loop if the desired value has been found
            break
        else:
            # return -1 for choice if the value is not found in the
list
            choice = -1


    return choice
#_____________________________________________________________________
_________________




# BEGIN MAIN PROGRAM #
#_____________________________________________________________________
_________________

# Make lists to hold our single run times and C values for our
algorithms later.
timeSeq = []
timeBin = []
timePyth = []

CSeq = []
CBin = []
CPyth = []
```

```
# Define a list of sample sizes we want to test.
sampleSize = [5000, 6000, 7000, 8000, 9000, 10000]

# Run the algorithms for a list of each sample size specified.
for value in sampleSize:
    unorderedList = []

    # Assign n random numbers to a list from 1 to n*n. Making the
random integer range large helps prevent duplicates.
    for i in range(value):

        unorderedList.append(randint(1, value*value))

    # We may have duplicates numbers next to each other in the list,
which may give inaccurate results for the algorithm.
    # So, we need to eliminate duplicates.
    # We can do this by converting the list into a set using set().
Sets are unordered collections that only accept unique values.
    # So, using set() will automatically remove duplicates.
    orderedSet = set(unorderedList)

    # Convert the set with removed duplicates back into a list again
    myList = list(orderedSet)

    # Sort the list
    myList.sort()

    listLength = len(myList)

    # If we did remove m duplicates from the list of size n, then our
new list size is n - m.
    # To give accurate results, let's re-add m terms randomly to the
list and re-sort it.
    # i.e. A list of 2000 values with 2 duplicates removed is now 1998
values,
    # so lets add 2 values back to the list and re-sort it to make a
new 2000 value'd list.
    while listLength < value:

        myList.append(randint(1, value))

        myList.sort()

        listLength = len(myList)


    # Being timing for sequential search
    timeStart1 = time()
    for rep1 in range(10000):
        seqSearch = sequential_search(myList, myList[value - 1])
```

```
    # End timing
    timeEnd1 = time()

    timeDiff1 = timeEnd1 - timeStart1

    timeSeq.append(format((timeDiff1/10000), ".3g"))

    # C = f(n) / n because sequential search is linear growth

    C1 = timeDiff1 / (10000 * value)
    # Format for 3 significant values
    CSeq.append(format(C1, ".3g"))



    # Being timing for binary search
    timeStart2 = time()
    for rep2 in range(10000):
        binSearch = binary_search(myList, myList[value - 1])
    # End timing
    timeEnd2 = time()

    timeDiff2 = timeEnd2 - timeStart2

    timeBin.append(format((timeDiff2/10000), ".3g"))



    # C = f(n)/log2(n) because the binary search is base 2 logarithmic
growth (f(n) = C*log2(n))
    C2 = timeDiff2 / (10000*log2(value))
    # Format for 3 significant values
    CBin.append(format(C2, ".3g"))

    # Being timing for Python built in search
    timeStart3 = time()
    for rep3 in range(10000):
        indPyth = myList.index(myList[value - 1])
    # End timing
    timeEnd3 = time()

    timeDiff3 = timeEnd3 - timeStart3

    timePyth.append(format((timeDiff3/10000), ".3g"))

    # C = f(n) / n because the python index search is linear growth
    C3 = timeDiff3 / (10000 * value)
    # Format for 3 significant values
    CPyth.append(format(C3, ".3g"))

    # put try or except code here
```

```
#_____________________________________________#
# Print the results to the user.
print("\n\n")

# Sequential Search Results
print("SAMPLE SIZES")
print(sampleSize)
print("\nRESULTS")
print("Sequential Search Results")
print("Times")
print(timeSeq)
print("Sequential Search C Values")
print(CSeq)

# Binary Search Results
print("\nBinary Search Results")
print("Times")
print(timeBin)
print("Binary Search C Values")
print(CBin)

# Python Built In Search Results
print("\nPython Built In Search Results")
print("Times")
print(timePyth)
print("Python Build In Search C Values")
print(CPyth)
```

MATLAB Code

```
% Set sample size vector
n = [5000, 6000, 7000, 8000, 9000, 10000];

# Set vectors for the results of each algorithm
exp_time_seq = [0.000566, 0.000666, 0.000792, 0.000882, 0.00104,
0.00116 ];
exp_time_bin=[0.00000751, 0.00000761, 0.00000761, 0.00000751,
0.00000851, 0.00000841];
exp_time_pyth=[0.0000885, 0.000117, 0.000126, 0.000144, 0.000166,
0.000185];

% Estimated theoretical C values based on an average of experimental
values.
C_seq = 0.000000113;
C_bin= 0.000000612;
C_pyth=0.0000000183;

% Theoretical Run Times
theory_seq = C_seq *n;
```

```
theory_bin= C_bin*log2(n);
theory_pyth= C_pyth*n;

% Plot all the trends
figure(1)
clf
plot( n, exp_time_seq, 'ro' )

hold on
plot( n, exp_time_bin, 'mo')
plot(n, exp_time_pyth, 'ko')
plot( n, theory_seq, 'y-' )
plot( n, theory_bin, 'g-' )
plot( n, theory_pyth, 'c-' )

hold off

% Make a legend for the trends
legend( ' experimental seq ', 'experimental bin', 'experimental pyth',
'theory seq', 'theory bin', 'theory pyth', 'Location', 'northwest' )

% Format axes and title
xlabel( ' list length n ' )
ylabel( 'run time f(n) in sec' )
title( 'Search and compare Algorithms  ' )

set( gcf, 'Color', [ 1 1 1 ] )

% Plot all the trends on a loglog plot.
figure(2)
clf
loglog( n, exp_time_seq, 'mo' )
hold on
loglog( n, exp_time_bin, 'ko' )
loglog( n, exp_time_pyth, 'ro' )
loglog( n, theory_seq, 'b-' )
loglog( n, theory_bin, 'k-' )
loglog( n, theory_pyth, 'g-' )
hold off
%set axis label
xlabel('log(n)')
ylabel('log(f(n))')
%set axis title
title('Search and compare Algorithms  log-log Axes')
%set axis legend

legend( ' experimental seq ', 'experimental bin', 'experimental pyth',
'theory seq', 'theory bin', 'theory pyth' ,'Location', 'southwest')

axis tight
```