CSE 1384

Binary Search (Recursive)

Lab #5

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 18th, 2017

Design

Function name: binary_search_recursive

Parameters: Value wanting to be found, list you want to look through, left index of the sublist you are searching through, right index of the sublist you're searching through.

Returns: index of the value you are looking for or -1 if the value is not found.

Algorithm:

Def binary_search_recursive(value, list, listLeftIndex, listRightIndex):

Calculate the middle index by taking the average of the left index and right index

Floor the result if it is not a whole number

if the left index is greater than the right index:

return -1 to indicate that no value has been found

if the value of the list at the middle index is equal to the value sought:

return the middle index to indicate the value has been found at that index

else if the value of the list at the middle index is greater than the value sought:

set the right index to the left of the middle index

call the binary_search_recursive function with the new indices to look at only the left half of the list

return the result of the new binary_search_recursive call

else:

set the left index to the right of the middle index

call the binary_search_recursive function with the new indices

return the result of the new binary_search_recursive call

Algorithm Analysis for Binary Recursive Search

How long does one function call take?

One function call takes the time it takes to chop the list and half (which is the same as saying the cumulative time it takes the compare numbers and then set new indices to look either to the left or right of the list).

How many function calls are there?

The number of function calls is equal to the number of times the list needs to be chopped in half to find the desired value.

What is n?

n is the size of the list we want to search through.

What is C?

C is the time it takes to chop the list in half. If comparing values in a list and setting index values to look left or right in the list all take constant times, then the overall time it takes to chop the list will be a constant time C.

What is f(n)?

Let a list be of size n = 8. We can chop this list 3 times before n = 1 and the list becomes unable to be chopped (1^{st} chop -> n = 4, 2^{nd} chop -> n = 2, 3^{rd} chop -> n = 1). $Log_2(8) = 3$, so a list of size n in general can be chopped $log_2(n)$ times. ***The total time is therefore $f(n) = C*log_2(n)$.***

What is O(n)?

O(n) is the worst case time it takes the algorithm to run. The worst case is when the algorithm has to chop the entire list $log_2(n)$ times. ***Therefore, $O(n)$ is $log_2(n)$.***

Testing

Test 1

```
[11, 23, 34, 46, 58, 69, 73, 87, 90, 102, 115, 127, 133, 149, 154, 168, 179, 183, 195, 205]

looking for  11

L 0 11
R 19 205
M 9 102

L 0 11
R 8 90
M 4 58

L 0 11
R 3 46
M 1 23

L 0 11
R 0 11
M 0 11

Returned index:  0
Expected index:  0
test passed: YES
```

Test 2

```
[11, 23, 34, 46, 58, 69, 73, 87, 90, 102, 115, 127, 133, 149, 154, 168, 179, 183, 195, 205]

looking for  205

L 0 11
R 19 205
M 9 102

L 10 115
R 19 205
M 14 154

L 15 168
R 19 205
M 17 183

L 18 195
R 19 205
M 18 195

L 19 205
R 19 205
M 19 205

Returned index:  19
Expected index:  19
test passed: YES
```

Test 3

```
[11, 23, 34, 46, 58, 69, 73, 87, 90, 102, 115, 127, 133, 149, 154, 168, 179, 183, 195, 205]

looking for  2

L 0 11
R 19 205
M 9 102

L 0 11
R 8 90
M 4 58

L 0 11
R 3 46
M 1 23

L 0 11
R 0 11
M 0 11

L 0   R -1   L/R crossed
```

Test 4

```
[11, 23, 34, 46, 58, 69, 73, 87, 90, 102, 115, 127, 133, 149, 154, 168, 179, 183, 195, 205]

looking for  300

L 0 11
R 19 205
M 9 102

L 10 115
R 19 205
M 14 154

L 15 168
R 19 205
M 17 183

L 18 195
R 19 205
M 18 195

L 19 205
R 19 205
M 19 205

L 20   R 19   L/R crossed

Returned index:  -1
Expected index:  -1
test passed: YES
```

Test 5

```
[11, 23, 34, 46, 58, 69, 73, 87, 90, 102, 115, 127, 133, 149, 154, 168, 179, 183, 195, 205]

looking for  106

L 0 11
R 19 205
M 9 102

L 10 115
R 19 205
M 14 154

L 10 115
R 13 149
M 11 127

L 10 115
R 10 115
M 10 115

L 10   R 9   L/R crossed

Returned index:  -1
Expected index:  -1
test passed: YES
```

Timing

For Binary Search Recursive

List length	Single search time in sec	Comparison Time or C estimate in sec
256	3.91E-6	4.75E-7
1024	4.86E-6	4.81E-7
4096	5.83E-6	5.05E-7
16384	6.86E-6	4.60E-7
65536	7.73E-6	5.01E-7
262144	8.72E-6	4.93E-7

Estimated C: 4.85E-7

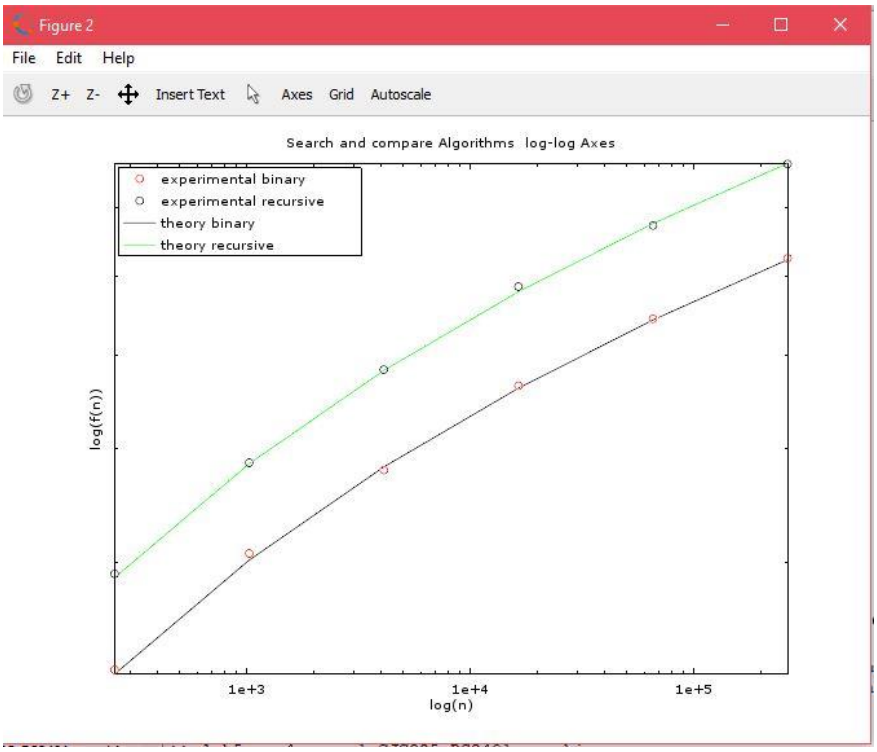
For Binary Search

List length	Single search time in sec	Comparison Time or C estimate in sec
256	3.24E-6	4.01E-7
1024	4.07E-6	4.07E-7
4096	4.79E-6	3.99E-7
16384	5.65E-6	4.00E-7
65536	6.44E-6	4.00E-7
262144	7.25E-6	4.21E-7

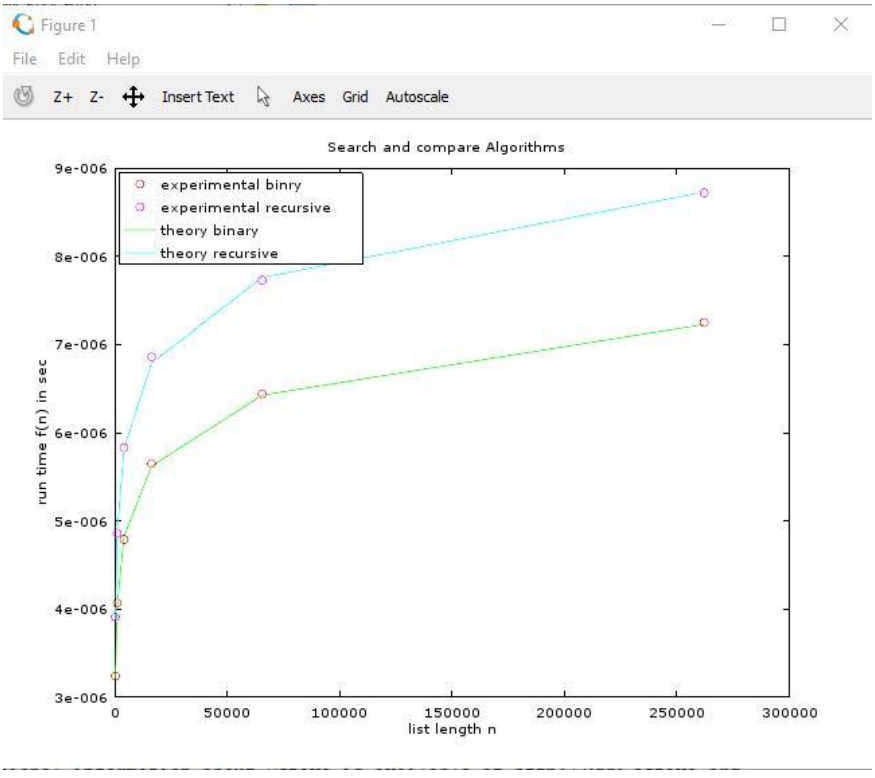
Estimated C : 4.016E-7

Graphing

log-log graph



Standard Graph (f(n) against n)



Conclusions

Question 1: According to your data, which is faster, looping or recursive binary search?

Looping binary search is faster based on the lower f(n) curves in the MatLab plots.

Question 2: What was the most challenging part of this assignment? How did you overcome the challenge?

Actually coding the algorithm was the toughest part of the assignment. The algorithm test cases would fail because the group would forget to return results after the function was called multiple times.

This was resolved by going over recursion notes and drawing diagrams of how the function gets called multiple times and returns values multiple times back to the original call.

Question 3: What helped you prepare effectively for this assignment? List three things. If you can't think of three, what will you do in the future to prepare effectively?

Recursion notes were reviewed, in-class recursion code was reviewed, and code from previous labs that was similar to code used in this lab was reviewed. With this in mind, the group had a familiarity with recursion before getting into the lab.

Code Appendix

```
# 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

from math import log2

from random import randint

# 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):

    # Calculate list length
    listLength = len(listVar)

    # Set left index and right index to the start and end of the list,
respectively.
    l_index = 0
    r_index = listLength - 1

    # While the indices have not crossed
    while l_index <= r_index:
```

```
        # Calculate the middle index and floor the result if it is not an
integer
        mid_index = int((l_index + r_index)/2)

        # Have we found the value we want with the middle index? Good, return
the middle index
        if listVar[mid_index] == numVar:

            choice = mid_index
            break
        # Is the value with the middle index bigger than the value we are
looking for? Start searching at the left half of the list
        elif listVar[mid_index] > numVar:

            r_index = mid_index - 1
        # Is the value with the middle index less than the value we are
looking for? Start searching at the right half of the list
        else:

            l_index = mid_index + 1

        # If the indices have crossed, return -1 because the value we want is
not in the list.
        if l_index > r_index:

            choice = -1

    return choice
#_____________________________________________________________________________
__________

# Function: binary_search_recursive(listVar, numVar, l_index, r_index)
# Purpose: Perform the binary search algorithm via recursion to find a value
in a list
# Parameters: listVar (The list to search through), numVar (The number to
look for), l_index (The left index position),
#             r_index (The right index position)
#_____________________________________________________________________________
def binary_search_recursive(listVar, numVar, l_index, r_index):

    # Calculate the middle index
    mid_index = int((l_index + r_index) / 2)


    # Have the left and right indices crossed? Return -1
    if l_index > r_index:
        choice = -1

        #print("L", l_index, "  R", r_index, "  L/R crossed")
        #print("")

        return choice

    #print("L", l_index, listVar[l_index])
    #print("R", r_index, listVar[r_index])
    #print("M", mid_index, listVar[mid_index])
    #print("")
```

```
    # Have we found the value we are looking for with the middle index?
    # Return the middle index
    if listVar[mid_index] == numVar:
        choice = mid_index
        return choice

    # Is the value we found with the middle index greater than the value
    # we are looking for?
    elif listVar[mid_index] > numVar:
        # Set the right index to the left of the middle index
        r_index = mid_index - 1

        # Call the function again to only look to the left side of the list
        # and return the result
        recurse1 = binary_search_recursive(listVar, numVar, l_index, r_index)
        return recurse1

    # Is the value we found with the middle index less than the value we are
    # looking for?
    else:
        # Set the left index to the right of the middle index
        l_index = mid_index + 1

        # Call the function again to only look to the right side of the list
        # and return the result
        recurse2 = binary_search_recursive(listVar, numVar, l_index, r_index)
        return recurse2

#_____________________________________________________________________________

# Define a sample list
testList = [11, 23, 34, 46, 58, 69, 73, 87, 90, 102, 115, 127, 133, 149, 154,
168, 179, 183, 195, 205]

# Starting position for right index. Indices start at 0, so we subtract one
from
# the list length
start_right = len(testList) - 1

# Test case for the first vale in the list
print(testList)
print("")
print("looking for ", 11)
print("")
result1 = binary_search_recursive(testList, 11, 0, start_right)
print("Returned index: ", result1)
print("Expected index: ", 0)

if result1 == 0:
    print("test passed: YES")
else:
    print("test passed: NO")
print("")
```

```
# Test case for the last value in the list
print(testList)
print("")
print("looking for ", 205)
print("")
result2 = binary_search_recursive(testList, 205, 0, start_right)
print("Returned index: ", result2)
print("Expected index: ", 19)

if result2 == 19:
    print("test passed: YES")
else:
    print("test passed: NO")
print("")

# Test case for a value less than the first value in the list
print(testList)
print("")
print("looking for ", 2)
print("")
result3 = binary_search_recursive(testList, 2, 0, start_right)
print("Returned index: ", result3)
print("Expected index: ", -1)

if result3 == -1:
    print("test passed: YES")
else:
    print("test passed: NO")
print("")

# Test case for a value bigger than the last value in the list
print(testList)
print("")
print("looking for ", 300)
print("")
result4 = binary_search_recursive(testList, 300, 0, start_right)
print("Returned index: ", result4)
print("Expected index: ", -1)

if result4 == -1:
    print("test passed: YES")
else:
    print("test passed: NO")
print("")

# Test case for a value not in the list but, if it was, it would be somewhere
in the middle.
print(testList)
print("")
print("looking for ", 106)
print("")
result5 = binary_search_recursive(testList, 106, 0, start_right)
print("Returned index: ", result5)
print("Expected index: ", -1)

if result5 == -1:
    print("test passed: YES")
```

```
else:
    print("test passed: NO")
print("")



# TIMING PORTION

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


CRec = []
CBin = []




# Define a list of sample sizes we want to test.
sampleSize = [256, 1024, 4096, 16384, 65536, 262144]

# 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, 10000000))

    # 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, 10000000))

        myList.sort()

        listLength = len(myList)



    # Begin timing for recursive binary search
    timeStart1 = time()
    for rep1 in range(900000):
        recSearch = binary_search_recursive(myList, myList[value - 1], 0,
(value - 1))
    # End timing
    timeEnd1 = time()

    timeDiff1 = timeEnd1 - timeStart1

    timeRec.append(format((timeDiff1/900000), ".3g"))

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

    C1 = timeDiff1 / (900000 * log2(value))
    # Format for 3 significant values
    CRec.append(format(C1, ".3g"))



    # Being timing for binary search
    timeStart2 = time()
    for rep2 in range(900000):
        binSearch = binary_search(myList, myList[value - 1])
    # End timing
    timeEnd2 = time()

    timeDiff2 = timeEnd2 - timeStart2

    timeBin.append(format((timeDiff2/900000), ".3g"))



    # C = f(n)/log2(n) because the binary search is base 2 logarithmic growth
(f(n) = C*log2(n))
    C2 = timeDiff2 / (900000*log2(value))
    # Format for 3 significant values
    CBin.append(format(C2, ".3g"))


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

# Binary Search Recursive Results
print("SAMPLE SIZES")
print(sampleSize)
```

```
print("\nRESULTS")
print("Binary Search Recursive Results")
print("Times")
print(timeRec)
print("Binary Search Recursive C Values")
print(CRec)

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

Matlab Code

```
% Set sample size vector
n = [256, 1024, 4096,16384,65536,262144   ];

# Set vectors for the results of each algorithm

exp_time_rec=[0.00000391,0.00000486,0.00000583,0.00000686,
0.00000773,0.00000872];
exp_time_bin=[0.00000324, 0.00000407, 0.00000479,
0.00000565,0.00000644,0.00000725   ];

% Estimated theoretical C values based on an average of experimental values.

C_bin=0.0000004016;
C_rec=0.000000485 ;

% Theoretical Run Times

theory_bin= C_bin*log2(n);
theory_rec= C_rec*log2(n);

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

hold on
plot( n, exp_time_rec, 'mo')
plot( n, theory_bin, 'g-' )
plot( n, theory_rec, 'c-' )

hold off

% Make a legend for the trends
legend( 'experimental binry', 'experimental recursive', 'theory binary',
'theory recursive', '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_bin, 'ro' )
hold on
loglog( n, exp_time_rec, 'ko' )
loglog( n, theory_bin, 'k-' )
loglog( n, theory_rec, '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 binary', 'experimental recursive','theory
binary','theory recursive','Location', 'northwest')

axis tight
```