Justin Pytel-Meyer

Aaron Tran-Huyhn

**CS 116 Fall 2022 Lab #09**

Due: **Wednesday, November 16, 2022 11:00 PM CST**

Points: **20**

**Instructions:**

1. Use this document template to report your answers and create separate java files for your classes. Enter all lab partner names at the top of first page.
2. You don’t need to finish your lab work during the corresponding lab session.
3. ZIP your Java files and lab report into a single file. Name the file as follows:

LastName\_FirstName\_CS116\_Lab9\_Report.zip

1. Submit the final document to Blackboard Assignments section before the due date. No late submissions will be accepted.
2. ALL lab partners need to submit a report, even if it is the same document.

**Objectives:**

1. (13 points) Understand the basics of sorting and runtime analysis,
2. (7 points) Algorithm time complexity problems

**Problem 1 [13 points]:**

For this assignment you are to implement and compare two different sorting methods on an array or an ArrayList A of integers.

First, implement insertion sort. Next, implement a version of insertion sort, called binary insertion sort, that locates where A[j] needs to be inserted into the sorted sequence A[1..j-1] by using binary search, and then performs the insert in the array or ArrayList.

|  |
| --- |
| **NOTE: Recall two facts from the lecture:**   * **binary search REQUIRES a sorted array,** * **in insertion sort the “left” subarray is always sorted.** |

1a. Before coding and running the trials, make a prediction (using Big O notation) for the growth in runtime of each sort.

|  |
| --- |
| **Your prediction:** |
| The insertion sort that does not use binary search will have O(n^2) time complexity while the insertion sort that **does** use binary search will have a O(nlogn) time complexity. This is because both versions will have to go every element at least once, making a baseline complexity of O(n), and then the binary search will reduce the searching for the correct spot of the current value to O(logn) as opposed to comparing every element, O(n) time complexity. |

1b. Write code and run the experiments:

* write a "main()" method to set up and run and time the sorts, and
* write a method for insertionSort, and
* write a method for binaryInsertionSort.

Run multiple trials (100 each) for the two sorts on a new set of random integers (between -10000 and 10000) of each size (1000, 2000, 4000, 8000, and 16,000 random integers).

Time (adapt the code provided below) the work done sorting by each algorithm on each size input array over the multiple trials of the same size to get the average runtime over the 100 trials. **You should only time the sorts, not the generating of new random numbers for each trial**. Use Excel or some other plotting software to graph the results for each sort of input size (x-axis) vs. average run time (y-axis).

1c. How do the experimental growth running times of the two sorts compare with your predictions?

|  |
| --- |
| **Your answer:** |
| Its hard to tell what that exact growth rate is but the binary sort is significantly faster at each N increment and grows far slower than the original insertion sort. |

1d. Does binary insertion improve the growth in running time of insertion sort? Discuss why or why not.

|  |
| --- |
| **Your answer:** |
| The difference even between the initial 1 thousand integer sorting is already quite large but the average sort time grows exponentially as the length of the array grows without the binary search. This implies that the binary search, which does not have explosive growth, significantly helps the run time. |

One way to measure run time in Java:

|  |
| --- |
| **Sample time measurement code:** |
| import java.util.concurrent.TimeUnit;  // Program to measure elapsed time in Java  class TimeUtil {  public static void main(String[] args) throws InterruptedException {  long startTime = System.nanoTime();  /\* ... the code being measured starts ... \*/  // sleep for 5 seconds  TimeUnit.SECONDS.sleep(5);  /\* ... the code being measured ends ... \*/  long endTime = System.nanoTime();  // get difference of two nanoTime values  long timeElapsed = endTime - startTime;  System.out.println("Execution time in nanoseconds : " + timeElapsed);  System.out.println("Execution time in milliseconds : " +  timeElapsed / 1000000);  }  } |

**Problem 2 [7 points]:**

**2a. (2 points)** Write pseudocode for an algorithm that locates an element in a list of increasing integers by successively splitting the list into three sublists of equal (or as close to equal as possible) size, and restricting the search to the appropriate piece (**so: each time the input size n changes! And you are working on a sub-problem – same approach, different input size**). It is possible that the element is not in the list. Also describe the worst-case time complexity, measured in terms of list element comparisons, of your algorithm.

**2b. (2 points)** Given a list S if n real numbers sorted in increasing order, and another real number x, design an efficient algorithm to determine whether or not there exist two distinct elements in S whose sum is exactly x. Also describe the worst-case time complexity, measured in terms of list element comparisons, of your algorithm.

**2c. (3 points)** In each of the following pseudocode segments, the integer variables i,j,n, and sum are declared earlier in the program. The value of n (a positive integer) is supplied by the user prior to execution of the segment. In each case we define the run-time function T(n) to be the number of times sum := sum + 1 is executed. Determine the best "big-O" form for T(n) for each pseudocode segment.

|  |  |  |
| --- | --- | --- |
| I **Time Complexity: 0(n^2)** | II **Time Complexity: 0(MN)** | III **Time Complexity: 0(logn)** |
| Begin  sum = 0  For i = 1 to n do  For j = 1 to n do  sum = sum + 1  End | Begin  sum = 0  For i = 1 to n do  For j = i to n do  sum = sum + 1  End | Begin  sum = 0  i = n  While i > 1 do  sum = sum + 1  i = i / 2  End |

**Answers:**

2.

a. The program would go as follows:

GnomeSort(int targetNum ,int[] inputArr)

Int size = Integer divide **starting array** to get three approximately equals portions

Int low = final index of first third of the array

Int mid = final index of second third of the array

Int high = final index of the array

While(size > 1){

if targetNum = inputArr[mid]:

Return mid

Else if targetNum = inputArr[high]:

Return high

Else if targetNum = inputArr[low]:

Return low

Else If targetNum < inputArr[low]:

High = low

Mid = 2/3 of size

Low = 1/3 of size

Else if targetNum < inputArr[mid]:

High = mid

Mid = low + 2/3 size

Low = low + 1/3 size

Else:

Low = mid + 1/3 size

Mid = mid + 2/3 size

Int size = Integer divide **previous** size of array to get three approximately equals portions

}

Reurn -1 (indicating that the element does not exist in the array)

**Time Complexity of a**: For every iteration of the while loop the method makes a 6 comparisons to the target value, the question is how many times does the while loop run. For this I will rely on a common knowledge of the binary search algorithim that runs in a general time complexity of O(logN) but more specifically runs in a log\_2(N) time complexity. Since this method does the same thing but with 3 partitions rather then 2 it would be safe to say that it has a time complexity of log\_3(N) that simplifies to O(logN) as N grows to a ridiculous size. Therefore O(logN) is the time complexity.

b. The method would be:

//For each element, as long as the current element plus the final element is

LeetCodeQuestion(int targetVal, int[] inArray){

Int low = 0;

Int high = array.length – 1;

while (low != high){

if(values at low and high summed is larger than target value)

high -= 1;

//Since the array is in sorted order, no combination with this high index will be smaller than the target value. Therefore we take the next smallest value to check

elif (values at low and high is smaller than the target value)

low += 1;

// Since the array is in sorted order, no combination with this low index value will be smaller with a new high value. Therefore the small value needs to be increased

elif (values at low and high equal the target value)

return True

}

Return false (since every combination has been tried)

}

**Time Complexity of b**: The worst case of this algortihim is checking against the target value 2N times, one for larger and one for smaller, where N is the amount of elements in the array. This simplifies to O(N) time complexity.

c.

The **first block of code** will have a growth time of O(N^2) due to the first variable **i** iterating over every element once and the second variable **j** doing the same but also invoking the sum iteration line everytime. This means that the executed N times for for each element, where N is the amount of elements, making the times sum is iterated N \* N times.

The **second block of code** is a little differnct with its O(MN) time complexity, it first goes every element once providing the O(N) part of the time complexity. Unlike the first block however it only goes over M elements where M is a seperate value from N due to it being always smaller than N. I have seen this complexity refered to as O(M\*N) and hope that this is acceptable as long as I explain what M is.

The **final block is a bit faster** with it being O(logN). The reason I say that it is logN is due to it accessing the middle point of each division of the provided elements and then adding to sum. This means that for each execution of the sum iteration line, it takes an exponential amount of increase in N to makes that change happen. Inversing that relation ship finds a O(logN) time complexity.