Thursday, 30 November 2023

Lab 2

Question 1. Comparing Algorithms. Problem: Find the THIRD largest in an array.

**Algorithm 1** : Idea – Use three loops one after another. First will find Max. Second will find Second Max,

Third will find third max. Note that it is possible First max == second Max == Third Max as in

7, 20, 18, 4, 20, 19, 20, 3.

and your program should return 20 in this case.

**Algorithm 2** : Idea – Use one loop. Maintain three variable max, preMax and prePreMax such that max

will have the maximum value, preMax will have the second largest and prePreMax will have the third

largest value.

In this lab, for both algorithms you will

(a) write the pseudo code. (Must follow the notations and conventions used in today’s Lecture)

(b) determine the worst-case time complexity by counting as in Slide 15 Lesson 2.

(c) Perform an empirical time comparison by implementing using Java similar to what you did in

W1D1.

(d) Draw a chart to compare algorithms.

**Algorithm algorithm1(array)**

Input: array of integers

Output: third largest value

if length(array) is 0:

return Integer.MIN\_VALUE;

maxIndex ← -1

secondMaxIndex ← -1

max ← Integer.MIN\_VALUE

for i ← 0 to length(array) - 1 do:

if array[i] > max then:

max ← array[i]

maxIndex ← i

max ← Integer.MIN\_VALUE

for i ← 0 to length(array) - 1 do:

if i ≠ maxIndex and array[i] > max then:

max ← array[i]

secondMaxIndex ← i

max ← Integer.MIN\_VALUE

for i ← 0 to length(array) - 1 do:

if i ≠ maxIndex and i ≠ secondMaxIndex and array[i] > max then:

max ← array[i]

return max

**Algorithm algorithm2(array)**

Input: array of integers

Output: third largest value

if length(array) is 0:

return Integer.MIN\_VALUE

max ← Integer.MIN\_VALUE

preMax ← Integer.MIN\_VALUE

prePreMax ← Integer.MIN\_VALUE

for i ← 0 to length(array) - 1 do:

number ← array[I]

if number > max then:

prePreMax ← preMax

preMax ← max

max ← number

else if number > preMax then:

prePreMax ← preMax

preMax ← number

else if number > prePreMax then:

prePreMax ← number

return prePreMax

b)

**Analyzing Algorithm1:**

There are 3 loops:

// n assignments of I

// n times of incrementing i

// n comparisons made on i along with traversing

for i ← 0 to length(array) - 1 do:

// n times array[i] has been accessed

// n comparisons made

if array[i] > max then:

// n times array[i] has been accessed

// n assignments of max

max ← array[i]

// n assignments of maxIndex

maxIndex ← i

Total for the loop: 8n

Other 2 loops are the same except they have 1 and 2 additional comparisons on each element respectively:

max ← Integer.MIN\_VALUE

for i ← 0 to length(array) - 1 do:

if i ≠ maxIndex and array[i] > max then:

max ← array[i]

secondMaxIndex ← I

Total: 9n since 1 more comparison (i ≠ maxIndex) was added

max ← Integer.MIN\_VALUE

for i ← 0 to length(array) - 1 do:

if i ≠ maxIndex and i ≠ secondMaxIndex and array[i] > max then:

max ← array[i]

Total: 9n since 1 more comparison (ii ≠ secondMaxIndex) was added but no index assignment in this loop

Grand Total Time complexity for the worst case of the algorithm is: 8n+9n+9n+c=26n where c represents number of constant operations

**Analyzing Algorithm 2:**

There is one loop:

// n assignments of I

// n times of incrementing i

// n comparisons made on i along with traversing

for i ← 0 to length(array) - 1 do:

// array[I] will be accessed n times

// n assignments

number ← array[I]

// n comparisons may be made

// 3n assignments may be made

if number > max then:

prePreMax ← preMax

preMax ← max

max ← number

// n-1 comparisons may be made

// 2(n-1) assignments may be made

else if number > preMax then:

prePreMax ← preMax

preMax ← number

// n-2 comparisons may be made

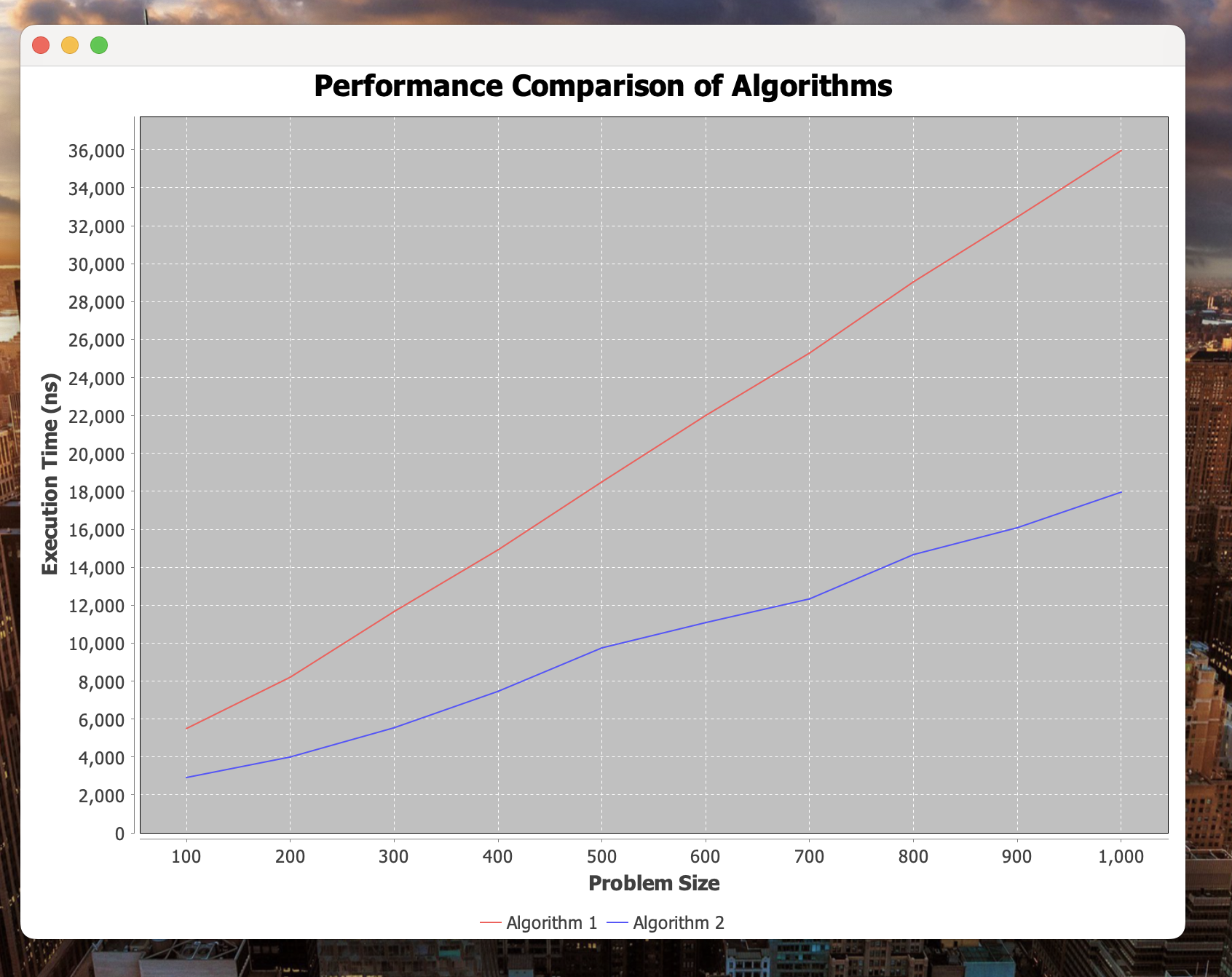
// n-2 assignments may be made

else if number > prePreMax then:

prePreMax ← number

return prePreMax

Grand Total Time complexity for the worst case of the algorithm is: 13n + c where c represents number of constant operations

c) You can find Algorithm.java file along with Main.java file. You can validate Algorithm.java with basic input included without any additional library. However, to run Main.java, you will need to install JFreeChart through maven since the library was used to draw the graph along with Swing. Although, the screenshot of the graph is available at d) below.

d) Comparison graph:

Neglecting constant operations, we could evaluate an approximate ratio of time complexities between algorithms as (26n)/(13n) = 2 which is also represented on the graph

Question 2. Consider the following functions to determine the relationships that exist among the complexity classes they belong.

|  |  |
| --- | --- |
| 10,1 | O(1) |
| log(logn) | O(log(logn)) |
| ln n | O(ln n) |
| logn | O(logn) |
| n^(1/k) (k>3) | O(n^(1/k) (k>3)) |
| n^(1/3) | O(n^(1/3)) |
| n^(1/2) | O(n^(1/2)) |
| n^(1/3)logn | O(n^(1/3)logn) |
| n^(1/2)logn | O(n^(1/2)logn) |
| nlogn, logn^n | O(nlogn) |
| n^2 | O(n^2) |
| n^3 | O(n^3) |
| n^k (k>3) | O(n^k) (k>3) |
| 2^n | O(2^n) |
| 3^n | O(3^n) |
| n! | O(n!) |
| n^n | O(n^n) |