CS 201, Fall 2021
Homework Assignment 2
Due: 23:59, December 05, 2021

Note: Before implementing this assignment please make sure that you understand the concept of time complexity analysis. In this assignment, we will perform the time complexity analysis of several array subset algorithms. That is, given two arrays `arr1` and `arr2` of size n and m, respectively, the algorithms find if the second array is a subset of the first one.

Note: You do not have to run this code on the server and you can use your own computer. This homework is about complexity analysis and the report is an important component.

Consider the following three algorithms. Each algorithm has different time complexity. The inputs to all algorithms will be the same. You can find some discussion on these algorithms (as well as some other alternatives) in the following link:
https://www.geeksforgeeks.org/find-whether-an-array-is-subset-of-another-array-set-1/
Note that if arr2 contains only unique items, all three algorithms produce the same result.

Algorithm 1 (Simple algorithm using linear search): Linear subset algorithm which works in O(n*m) time, where n and m are the sizes of the arrays. You can use nested loops where the outer loop selects each element of arr2 one by one and the inner loop linearly searches for these elements in arr1. If all elements of arr2 are found in arr1, return 1; otherwise, return 0.

Algorithm 2 (Using sorted arrays with binary search): Given two sorted arrays arr1 and arr2, this algorithm uses binary search to search for each item of arr2 in arr1. For this assignment, sorting time is not included in the time required for the subset algorithm. Therefore, the complexity of this algorithm is O(m*logn).

Algorithm 3: (Using frequency tables):

- Create a frequency table for the elements of arr1. For this, use an array whose size is the maximum value in arr1. You can find this value by making a linear pass over arr1. Once this array is constructed with all values initialized to 0, you can count the frequency of each unique element in arr1 in O(n) time.
- Traverse arr2 and search for each element of arr2 in the frequency table of arr1. If the element is found in arr1, decrease the frequency by one. If at least one element of arr2 is not found, return 0.
- If all elements of arr2 are found in arr1, return 1.

The complexity of this algorithm is O(n+m).

ASSIGNMENT:

Answer each of these questions:

1. Study the algorithms and understand their upper bounds. Describe how the complexity of each algorithm is calculated in your own words.
2. Report parameters of the computer that you used. Report RAM and Processor specifications in particular.

Perform these operations:

3. Create an implementation of all of these functions as global functions in your main file. Then in your main function call these functions to measure their running time. For simplicity, you can use a sorted array for arr1 in doing the analysis. More specifically, do the following: Pick a range for the elements that will be inserted into the arrays. Create a sorted arr1 of size n and a separate array arr2 (can be unsorted with random numbers in the same range) of size m. Make sure that arr2 contains unique items. The time required for constructing these arrays is not included in the complexity analysis. Then, test all three methods using the same exact arrays. For fair comparison do not alter any array before you try it with all three algorithms. Create a table containing all the values that you have recorded and include it in your report. The layout of the table is shown below. You cannot change the column format of the table but you can include as many rows as you like as long as the total number of rows (different values of n that you used) is more than or equal to 10. This means that you have to try at least 10 different n values. Your choice of n values should be adjusted so that you can observe the expected time complexities in the actual runs.

n	Algorithm 1		Algorithm 2		Algorithm 3	
	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$	$m = 10^3$	$m = 10^4$
10^5						
$2 * 10^5$						
$3 * 10^5$						
$4 * 10^5$						
$5 * 10^5$						

Observe the organization of the table. The rows of the table represent all the different numbers of n values that we will try. In the example table above, we used values 10^5, $2 * 10^5$,... which have a difference of 10^5 between them. You do not have to use the same exact values or the same interval. Determine values and intervals that give you the best plot on your computer. But make sure that values are evenly spread from each other. Also for demonstration purposes, we used only 5 different n values, while you have to use at least 10 different n values. As for the

range of the values, make sure that it is sufficiently large so that your function call with the largest n value takes at least 1 minute.

4. Create a plot for each column of the table (total of 6) , where n values are on the x-axis and elapsed times are on the y axis. Note that you are responsible for the quality of your plots. Sloppy and poorly scaled plots will lead to a reduction of a significant amount of points even if you conducted all the experiments perfectly. Consider using Python or MATLAB for plotting, as they contain built-in libraries for plotting similar functions. If the plot you generated makes it hard to draw conclusions about the behavior of the algorithm, then you should probably consider another way of plotting it. Name your plots properly. So that we can differentiate which one is which. None that here you are simulating the asymptotic behavior of the function, and the plots you generate are not expected to look like perfect logarithmic or linear plots. There will be some fluctuations, especially with smaller n values, therefore it is all the more important to pick the proper range of n values.

You can use the following code segment to compute the execution time of a code block. For these operations, you must include the `ctime` header file.

```
double duration;
clock_t startTime = clock();
//Code block
//...
//Compute the number of seconds that passed since the starting time
duration = 1000 * double( clock() - startTime ) / CLOCKS_PER_SEC;
cout << "Execution took " << duration << " milliseconds." << endl;
```

An alternative code segment to compute the execution time is as follows. For these operations, you must include the `chrono` header file.

```
//Declare necessary variables
std::chrono::time_point< std::chrono::system_clock > startTime;
std::chrono::duration< double, milli > elapsedTime;
//Store the starting time
startTime = std::chrono::system_clock::now();
//Code block
...
//Compute the number of seconds that passed since the starting time
elapsedTime = std::chrono::system_clock::now() - startTime;
cout << "Execution took " << elapsedTime.count() << " milliseconds." << endl;
```

NOTES ABOUT SUBMISSION:

1. This assignment will be graded by your TA Aydamir Mirzayev (aydamir.mirzayev@bilkent.edu.tr). Thus, you should ask your homework-related questions directly to him.
2. The assignment is due by December 05, 2021, at 23:59. You should upload your homework to the upload link on Moodle before the deadline. No hardcopy submission is needed. The standard rules about late homework submissions apply. Please see the course web page for further discussion of the late homework policy as well as academic integrity.
3. You must submit a report (as a PDF file) that contains all information requested above (plots, tables, computer specification, discussion) and a cpp file that contains the main function that you used as the driver in your simulations as well as the solution functions in a single archive file.
4. You should prepare your report (plots, tables, computer specification, discussion) using a word processor (in other words, do not submit images of handwritten answers) and convert it to a PDF file. Handwritten answers and drawings will not be accepted. In addition, DO NOT submit your report in any other format than .pdf.
5. The code segments given above may display 0 milliseconds when the value of n is small. Of course, the running time cannot be 0 but it seems to be 0 because of the precision of the used clock. However, we will not accept 0 as an answer. Thus, to obtain a running time greater than 0, please consider running algorithm k times using a loop, and then divide the running time (which will be greater than 0) by k.
6. You can reuse the codes on the lecture slides if you need, but other than that, the code must be your own implementation.