CS 201 - Fall 2023

Homework 2: Algorithm Efficiency and Sorting

Due: Nov 12, 2023, 23:59

Introduction:

Bilkent Library asks for our help!

The library's collection is vast and diverse, with a multitude of digital artifacts. To ensure efficient access to these resources, the library's data management system should be in its most efficient state. One critical aspect of reaching this objective is making use of a performant sorting algorithm and the library wants our advice in this regard.

Objective:

Your mission is to explore the complexities of sorting algorithms to help the digital library enhance its search and retrieval capabilities. In this quest, you will compare given algorithms from the perspectives of ***memory*** and ***computational*** complexity.

The library uses integers to identify each book within its application. Thus, you can consider that the book collection is represented as an array of ***integers***.

Please keep in mind that we expect you to support your deductions with experiment results.

Task 1: Which Algorithm? – 70 points

We suggested the library use either **Merge Sort** or **Quick Sort**, but one of the library's staff insists on using **Bubble Sort**. Now, we are asked to give them a clear suggestion and justification about using which algorithm. First, you have to implement those algorithms. Then, you have to analyze their memory and time complexities. You must perform experiments considering various situations with varying sized arrays: Random Array, Sorted Array (Descending), and Sorted Array (Ascending).

To measure the time complexity, you can count each book comparison your algorithm performs. That is, consider "k1 < k2" as one comparison, where k1 and k2 correspond to book IDs in the collection.

To measure the memory complexity, you must measure the total memory dynamically allocated by each algorithm. For this, you can sum up the number of dynamically allocated Book IDs.

After completing your experiments and analyzing the data, write a clear discussion regarding which sorting algorithm is more suitable for the library's specific needs and state your conclusion. Consider both memory and time complexity and provide well-reasoned recommendations.

Note: *We expect you to discuss both the theoretical and empirical complexities of these algorithms in a well-written two-page report.*

Task 2: New Insights – 30 points

The library authorities noticed that their data are almost sorted. They are wondering if having this assumption makes any difference in the best fitting algorithm. Now based on this information you have to suggest them an algorithm and support your suggestion with experimental results.

Note: *We expect to have a well-written one-page report for this task. In addition, you can have some image or table attachments.*

Important Notes

- This homework is prepared by your **TA Masoud Poorghaffar <m.poorghaffar@bilkent.edu.tr>**. Please direct your questions regarding this homework to him.
- The assignment is due by ***Nov 12, 2023, 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 NO late homework policy as well as academic integrity.
- Study algorithms and understand how the upper bounds are found for each. Describe how the complexity of each algorithm is calculated in your own words.
- 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. You do not need to run your implementation on the remote server for this homework. Run your code on your local machine and mention the specifications of your machine in your report.
- Any algorithm that you mention or discuss in your report must be implemented by you personally. This means that you are responsible for writing the code for these algorithms, and they should not be copied or sourced from external references or sources. You can reuse the codes from the slides.
- 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.
- You should experiment with each algorithm using at least 10 arrays of varying sizes. It is mandatory to include arrays of size 2^6, 2^9 and 2^{12}. The array with a size of 2^{12} should be considered as a large array for your testing purposes. Repeat every experiment 10 times and get the average values.
- For each array type (e.g., almost sorted), create two tables like below that show the (1) time and (2) memory consumed by the algorithms (columns) for varying input size (rows) as follows. (Overall, 6 tables for task 1 and 2 for task 2)

n	Bubble Sort	Merge Sort	Quick Sort
2^6			
2^9			
2^{12}			

- Generate a separate plot for each task (Part 1 and Part 2) and for each array type. Plot the "n" values on the x-axis and the corresponding measured values (time or memory) on the y-axis.
- You should prepare your report (plots, tables, computer specification, discussion) using a word processor and convert it into a PDF file. In other words, do not submit images of handwritten answers. Handwritten answers and drawings will not be accepted. In addition, DO NOT submit your report in any other format than pdf.
- We define an "almost sorted array" as an array in which approximately 10% of its elements are not in the correctly sorted order. You can generate random or almost sorted arrays using following functions:

```cpp
#include <cstdlib>
#include <iostream>

int *generate_random_array(int size) {
    // TODO: you have to implement this
}

int *generate_almost_sorted_array(int size) {
    srand((unsigned int)time(NULL));

    int *my_array = new int[size];
    for (int i = 0; i < size; i++) {
        my_array[i] = i;

        if (std::rand() % 10 == 1) {
            my_array[i] = i + size;
        }
    }

    return my_array;
}
```

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

```cpp
#include "timing.h"
#include <ctime>

double total_time = 0.0;
clock_t c_start = clock();

// run your sorting algorithm

clock_t c_end = clock();

double duration = (float) (c_end - c_start) / CLOCKS_PER_SEC;
cout << "Execution took " << duration << " seconds." << endl
```

Good Luck