ELEC 230 C++ Exercise (2021-2022)

I. Overview of a Sorting Algorithm

In the field of computer science, a sorting algorithm is a type of algorithm that puts elements of a list in a specific order. The most frequently used orders are numerical order and lexicographical (i.e., dictionary) order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output. More formally, the output of any sorting algorithm must satisfy two conditions:

- The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
- The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

Further, the input data is often stored in an array[1], which allows random access, rather than a list, which only allows sequential access; though many algorithms can be applied to either type of data after suitable modification.

Sorting algorithms are often referred to as a word followed by the word "sort" and grammatically are used in English as noun phrases, for example in the sentence, "it is inefficient to use insertion sort on large lists" the phrase insertion sort refers to the insertion sort sorting algorithm.

II. Your Exercise

Your goal will be to design and build a single-thread and multi-thread execution of a well-known sorting algorithm, called mergesort. You will use C++ to craft your code and time the execution for both single-thread and multi-thread implementation. A final step is to compare these two approaches and assess the performance of single and multi-threaded application.

The description and the requirements of your code is given in the section "**Rules and Requirements of your Code**".

[1] Throughout this assessment, the word array and vector are interchangeable if not stated otherwise.

III. Introduction & History

Merge Sort (also commonly spelled as mergesort in literature) is a well-known "divide and conquer" algorithm for sorting the elements in an array. This algorithm was invented by John von Neumann in 1945 [1] with a description and analysis of bottom-up sorting appeared in a report by Goldstine and von Neumann in 1948 [2].

IV. Operation Principle

During the divide phase, it recursively splits the input array into two halves referred to as the left half (L), and the right half (R). From here on, the merge phase repeatedly merges those sorted sub arrays to produce new larger sorted sub arrays. This process continues until there's only one sub array remaining, which is the final sorted result.

In a nutshell, a merge sort works as follows:

- Divide the unsorted list into **n** sublists, where each of these sublists contain one element (a list of one element is considered sorted).
- Repeatedly merge these sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

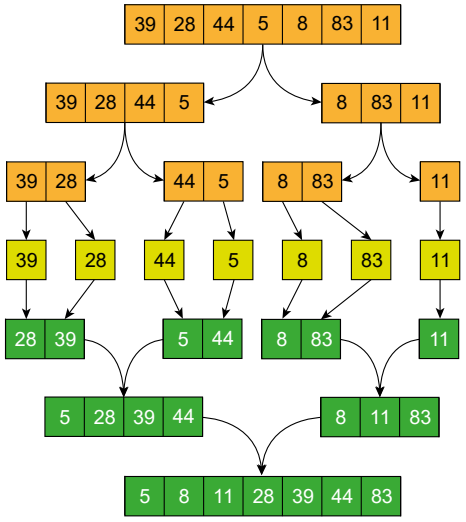

Figure 1. A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

V. Strategy

Using the Divide and Conquer method, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem. Assume we had to sort an array/vector **A**. A subproblem would be to sort a sub-section of this array/vector starting at index *p* and ending at index *r*, denoted as **A** [*p*…*r*].

1. Divide

If *q* is the half-way point between *p* and *r*, then we can split the subarray[2] **A** [*p*…*r*] into two arrays/vector **A** [*p*…*q*] (left) and **A** [*q*+*1*, *r*] (right).

[2] Throughout this exercise, the words subarray and subvector are interchangeable if not stated otherwise.

2. Conquer

In the conquer step, we try to sort both the subarrays **A** [*p*...*q*] and **A** [*q*+*1*, *r*]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

3. Combine

When the conquer step reaches the base step and we get two sorted subarrays **A** [*p*...*q*] and **A** [*q*+*1*, *r*] for array **A** [*p*...*r*], we combine the results by creating a sorted array **A** [*p*...*r*] from two sorted subarrays **A** [*p*...*q*] and **A** [*q*+*1*, *r*].

used to express idea between programmers using different language

VI. The Pseudocode for the Algorithm

Pseudocode is a plain language description of the steps in an algorithm or another system. Pseudocode often uses structural conventions of a normal programming language, but is intended for human reading rather than machine reading. The pseudocode for the top-down approach (bottom-up implementation also exists) is presented as follows;

```
function merge_sort(list m) is
    // Base case. A list of zero or one elements is sorted, by
definition.
    if length of m ≤ 1 then
        return m

    // Recursive case. First, divide the list into equal-sized sublists
    // consisting of the first half and second half of the list.
    // This assumes lists start at index 0.
    var left := empty list
    var right := empty list
    for each x with index i in m do
        if i < (length of m)/2 then
            add x to left
        else
            add x to right

    // Recursively sort both sublists.
    left := merge_sort(left)
    right := merge_sort(right)

    // Then merge the now-sorted sublists.
    return merge(left, right)

function merge(left, right) is
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
```

```
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

Rules and Requirements of your Code

You code must be executable in a Linux environment without any modification

Tip: It would be easier if you write the code in Linux as the use of **threads** may make portability harder without significant modification.

PART I – Single

In this section you are to devise this mergesort algorithm to work on a single thread. The following processes **must** to be implemented.

- The user will be able to enter an **n** number of values of their choice.
  - You are allowed to use either **vectors** or **arrays** for this approach. However, if you are to use arrays instead of vectors you need to pay attention to **pointers** and need work on **dynamic arrays**. (Both approaches are valid)
  - The user needs to be able to enter both **integer** values and **double** values.
- The user will be able to use a **file** containing random numbers.
  - This will be provided in CANVAS.
  - **Note:** This file will contain exactly 100,000 lines of random numbers generated within the range of 1 to 65535.
- Once an **n** size **vector** (or **array** if you prefer) is entered, the user needs to be informed of their values and therefore the **vector/array** of the entered values will be printed at the output.
  - If the user has used a file, only a text output of "file read" will be presented.
- After printing the **vector/array** containing the entered values, the **vector/array** needs to be checked.
  - If the **vector/array** is not sorted the vector/array is not operated upon in this step.
  - If the user entered an already sorted array, the **vector/array** needs to be reshuffled and checked again.
    - This checking-reshuffling is done until the **vector/array** is no longer sorted.
      - You must overcome the bug where the user enters just 1 value.
    - You are allowed to use any method you see fit to achieve this that has been taught at this module.
- The next step is to devise the **mergesort** algorithm.
  - The operation of this function cannot be written inside the main function and needs to be constructed outside.
    - You are allowed to use either functions or classes for this approach.

        - Every step of the mergesort must be printed to the output. The only exception is when the user uses a file where you must not show the steps.
- Once the operation is successful the user needs to be presented with their sorted list.
    - This is only done if the user enters the values themselves. Any other option, you are not to print the output.
- You are to print the execution time for your merge-sort algorithm.
    - **Tip:** You need to use the **<chrono>** header.

PART II – Multi Thread

In this part of the exercise you are to reuse the previous code to utilize multi-thread and to decrease the time it takes.

- **Tip:** You need to use the **<thread>** header.
- You are to print the execution time for your merge-sort algorithm.
    - **Tip:** You need to use the **<chrono>** header.

PART III

The last part of the Exercise is to compare these two algorithms.

- Instead of user input your algorithm needs to generate 100,000 random numbers for single threaded and multi-threaded application and the algorithm needs to run 100 times and the average execution time needs to be calculated.
    - **Comparison:** Which one ran faster? If so why?
- Instead of user input your algorithm needs to generate 100 random numbers for single threaded and multi-threaded application and the algorithm needs to run 100 times and the average execution time needs to be calculated.
    - **Comparison:** Which one ran faster? If so why?

Additional Requirements

Your code must not be longer than 250 lines.

- A new line is to be generated after each semicolon, unless the semicolon is in a statement (i.e., for loop while loop).
    - If you are using CLion, you can automate this by clicking Code > Reformat Code.
    - If you are using another IDE (i.e., VS, Codeblocks) please refers to it's documentation whether a similar feature is available.
- You are to use no more than 8 headers (i.e., #include …. lines)
- Your execution time measurement must start before you invoke your mergesort algorithm and must conclude when your algorithm is finished.
- All the abovementioned tasks must be written under one .cpp file.

References

[1]. Knuth, Donald E. *The art of computer programming: Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1998.

[2]. Katajainen, Jyrki, and Jesper Larsson Träff. "*A meticulous analysis of mergesort programs*." Italian Conference on Algorithms and Complexity. Springer, Berlin, Heidelberg, 1997.