COMPSCI **2XC3** — **Lab** 4

[Due Sunday, Feb 13th, 11:59pm]

Submit the lab on Avenue. Note, Avenue must receive your lab by the due date. **Do not leave your submission to the last minute! Late submissions, even by seconds, will not be graded.** You have been warned.

This lab is worth (59/7)% of your final grade in the course. Read this document carefully and completely. **Whenever I ask you to discuss something, I implicitly mean to include that discussion in your lab report.** Include all timing experiments in your `code.py` file.

Purpose

The goals of this lab are as follows:

1. Implement bottom-up and three-way mergesort.
2. Empirically analyze the performance of your implementations.

Submission

You will submit your lab via Avenue. You will submit your lab as three separate files:

- code.py
- sorts.py
- report.pdf (or docx, etc.)

Only one member of your lab group will submit to Avenue – see the section below for more details on that. Your report .pdf will contain all information regarding your group members, i.e. name, students number, McMaster email, and enrolled lab section. This will be given on the title page of the report. For the remainder of this document, read carefully to gauge what other material is required in the report. Your report should be professional and free from egregious grammar, spelling, and formatting errors. Moreover, all graphs/figures in your report should be professional and clear. You may lose grades if this is not the case. Organize the report in a such a way to make things easy for the TA to grade. You are not doing yourself any favors by making your report difficult to mark!

Mergesort [100%]

I am assuming you have seen mergesort in your other courses. Otherwise, do some independent research on what mergesort is. Understanding how it works is relatively straight forward.

Bottom-up [60%]

Everyone that learns mergesort has seen an image similar to the one below.

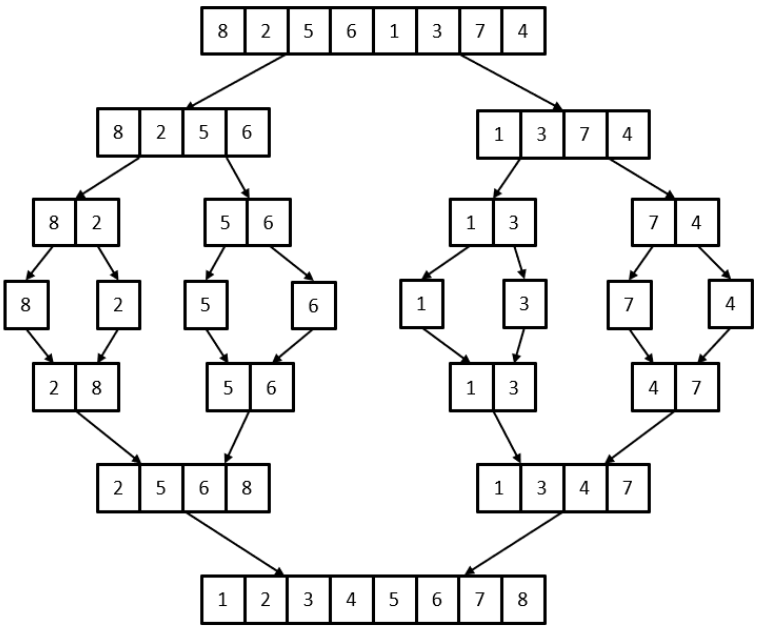

Figure 1: Classic top-down mergesort illustration.

Mergesort seems to embody the divide-and-conquer approach. But do we really need to divide? How about just conquering? What I mean by that is, instead of recursively splitting the list into smaller parts (from the top down), and then rebuild it. Why not instead start off by viewing the array/list as already divided, and simply complete the rebuild/merge portion. This is the essence of the bottom-up implementation. The idea is you iteratively pass through the list and merge portions of the list based off some window size (see the figure below). The size of the window increases after each iteration of the loop.

The bottom-up implementation does not make any recursive calls. It is done entirely with loops. Implement merge sort via a bottom-up approach. Meet the following criteria to achieve full grades:

- Implement a function `mergesort_bottom(L)` which takes a single list as input and sorts that list L.

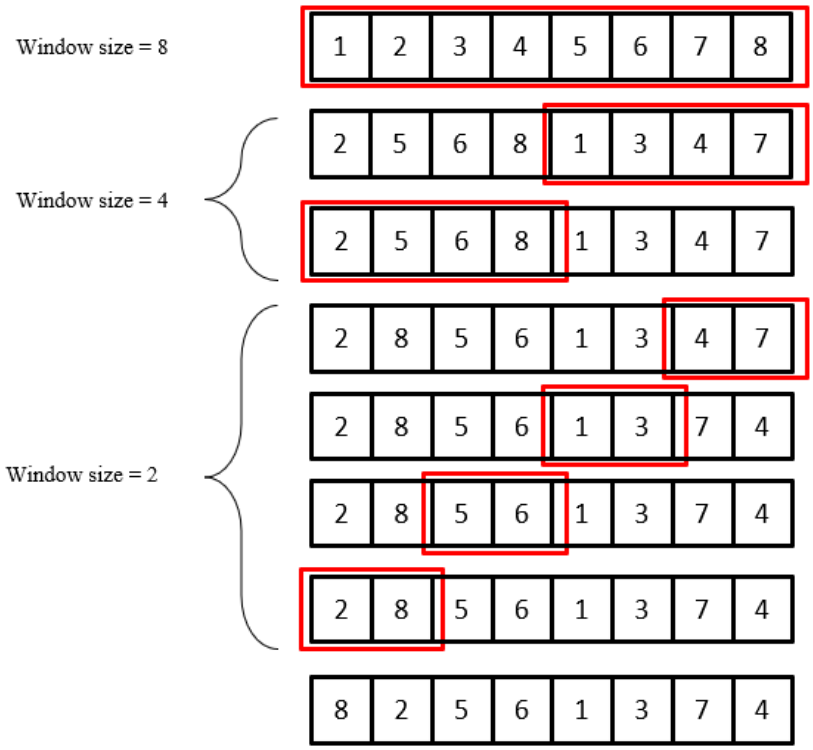

Figure 2: A bottom-up illustration of merge sort. Note the elements being sorted/merged in the window.

- Implement a helper function `merge_bottom(L, start, mid, end)`. This should merge the interval `[start:end]` assuming `[start:mid]` and `[mid:end]` are sorted. The exact inclusive/exclusive decisions for the intervals are left up to you. As long as your `mergesort_bottom(L)` works.
- Your functions should not be recursive in anyway. Loops only.
- Note, in Figure 2 the length of the list is conveniently a power of 2, this may not be the case. You will have to resolve this in some way.

Compare this bottom-up implementation to the one I gave you in `lab4.py`

Three-Way Mergesort [20%]

Traditionally, mergesort divides the list recursively in 2, but why not 3? Do you think this would be better or worse than a 2 way mergesort? Implement a three-way mergesort via `mergesort_three` and `merge_three` function in your `sorts.py` file. These functions should

be written in the same fashion as `mergesort` and `merge`. Compare the performance of the three-way mergesort vs the traditional two-way.

Worst Case [20%]

Last lab we saw that quicksort has a terrible worst case runtime when the list is sorted or near sorted. Is the same true for mergesort? Take your best mergesort implementation. Using the `create_near_sorted_list` function graph the runtime of mergesort vs *factor*, for factor between 0 and 0.5. Keep the length of the list constant throughout your experiments. What do you observe?