CS 271 Project 1 – Kinda Sorta (Sorting Algorithm Implementation)

Instructor: Flannery Currin

Due: September 17th, 2025 by 9:00 AM

1 Learning Goals

As computer scientists, you will often need to translate an algorithm into an implementation in some programming language. This process takes some practice, but it can also help deepen your understanding of the algorithms being implemented. In this project, you will implement several sorting algorithms and connect the runtime analysis we are doing in class and through the readings to real, concrete tests of your methods' runtimes. Specifically, after working on this project, you should:

- Understand how the selection sort, insertion sort, mergesort, and quicksort algorithms work and be able to trace through their execution
- Be comfortable and confident in translating a pseudocode algorithm into C++
- Know the relative strengths and weaknesses of these algorithms in terms of time and space efficiency
- Be able to identify and generate examples of best and worst case inputs for each of these algorithms
- Compare the runtime analyses we have done in class to the actual runtime of your sorting algorithm implementations and use that to identify and correct disconnects between the algorithm and your implementation

2 Project Overview

You may complete this project individually or with a partner. I will leave the pairing to you if you choose to work with a partner. Whether you work individually or in a pair, you are individually responsible for the learning goals above (dividing and conquering may not be the most effective strategy here). You may discuss this assignment with your partner, the course TA or instructor, but the work you submit must be your group members' own work. You can (and should) use your Project 0 as a starting point. You may consult sources like visualgo as references, but note that these sources do not necessarily align perfectly with the CLRS algorithms.

Extend your DoublyLinkedList class to implement the ability to sort the elements in list objects using selection sort, insertion sort, mergesort, and quicksort (CLRS chapter 7.1). After sorting a list object, the elements contained in its nodes should be in monotonically increasing order (smallest value in the head Node, largest value in the tail Node). To do this:

- Add four public method declarations to `DoublyLinkedList.h` called `selectionSort`, `insertionSort`, `mergesort`, and `quicksort`. Each of these should be void and take no parameters.
- Implement these methods in `DoublyLinkedList.cpp`.
- Declare and implement private helper methods as needed (swap, merge, partition, etc.). There is flexibility in how you design your class here.

As you implement each sorting algorithm, test it. To do this:

- Create a file `sorting_test.cpp`.
- Write unit tests like you did for Project 0. Any reasonable decomposition and naming for your unit test functions is fine. These tests should examine:
  - Correctness (are lists monotonically increasing as expected after calling the method? do they contain the same values as they did before sorting?)
  - Efficiency (is the order of growth what we would expect?)
- Create a PDF report as you test. For each algorithm:
  - Describe how you confirmed it produced correct output.
  - Describe how its runtime changes based on the size of the list and other characteristics of the data. Does this match what you would expect for this algorithm?
  - Describe any issues you ran into implementing this algorithm and how you approach resolving those issues.
- To help test efficiency, I have included three folders containing `.in` files containing integers. One folder contains `.in` files containing already sorted values; one contains files containing values sorted in reverse order; and one contains files containing randomly ordered values. Each folder contains `.in` files with different numbers of values (10, 100, 1000, etc.). For each of these files: read the values in and append them into a DoublyLinkedList and clock how long it takes to sort the list using each of your methods (follow the examples from project 0 for this).

- For each sorting algorithm and type of input (sorted, reversed, random), plot these runtimes (input size on x-axis and runtime on y-axis). You may create these plots however you would like, but each plot should be clearly titled with the algorithm and case (sorted, reversed, random). Later in the document, I provide an example of what one of these plots should look like and an option for creating these programmatically. Include these plots in your testing report.
- Create your own test case files to test your methods more comprehensively (using values other than integers, other patterns for the original ordering, etc.). Use these in your unit tests and report as well.

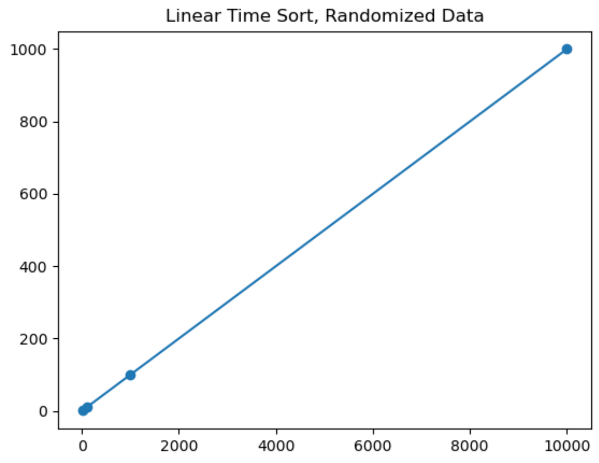
3 Project Submission

On Canvas, before the deadline, you will submit **a zip** containing:

- Your updated `DoublyLinkedList.h`.
- Your updated `DoublyLinkedList.cpp` with your method implementations.
- `sorting_test.cpp` with your unit tests (**remove plotting code/dependencies that require installs before submitting** – just output the results of of timed tests as text).
- Any additional test files you created (not the ones I included in the assignment), organized in whatever folder structure required for `sorting_test.cpp`.
- A PDF of your testing report.

4 Tips

- Begin early, focusing on **one algorithm at a time**. Write the declaration, implement it, test it, create plots, and write up that section of the report before moving to the next algorithm.
- The VSCode C++ debugger is your friend in this class. You will need to add the `-g` option when you compile to use the debugger. Totally acceptable to use external resources/copilot to work through setting the debugger up.
- To produce plots directly from your test file, consider using a library like Matplotlib for C++ or sciplot. These can be finicky to set up and learn to use, so it is also completely acceptable to output test results to a file and plug them into an external plotting tool. Below is an example of what a plot might look like for a hypothetical algorithm that can sort a list in linear time. **You may need to adjust the x and/or y limits to**



rescale your plots. Use dots or other markers to show each recorded data point. If you cannot see each individual point, you may need to re-scale your axes (the example below just barely shows the distinct n = 10 and n = 100 points, so it could potentially benefit from rescaling).

5 Grading Scheme

The out-of-class submission will be graded using the following scheme:

Criteria	**Description**	**Points**
Completeness	Meets submission requirements (files, documentation, etc.)	2
Correctness	Passes Dr. Currin's extended tests and follows specs	4
Report	Contains required components and thoughtful analysis	2
Testing	Tests should cover a range of types and cases, and logic should be sound	2
Total		**10**

The in-class component is worth 20 points. You should not need to study extra or memorize your out-of-class work for the in-class component. Working on the out-of-class component with the learning goals in mind is your best preparation. The in-class project component lets you test how prepared you are to apply the concepts covered by the project to a new scenario.