# Programming Assignment 2 - Sorting and Analysis

## Due: March 4

**This is a group assignment.** You may work individually or with one or two other students. The work scales according to the size of the group for 1, 2, or 3 (max) members.

You and your team are responsible for adhering to the Aggie Honor Code and student rules regarding original work. You and your team must write and submit your own code to Mimir and report to Canvas. Plagiarism checks will be used in addition to a manual review from the grader.

That said, we recognize that sorting is a well-known task in computer science, and it will be difficult to even search for references without coming across existing implementations. You can use references to help understand the problem, and you should also cite the materials you reference in the report. It is your responsibility to use references appropriately! *Do not copy code!*

## Overview

Searching and sorting are two of the most important concepts in the field of Computer Science. These activities enable countless applications, ranging from simple document processing (find & replace) to advanced artificial intelligence problem solving (constraint satisfaction searches) and more. We will see much more about searching and sorting in labs and upcoming lectures as these are the key activities that motivate many of the data structures and algorithms we will cover.

### *Code Deliverables (Mimir) - 50%*

#### SortLib

For this assignment, you will implement a templated C++ library that includes some well-known sorting algorithms. The library, `SortLib`, will consist of the following (based on group size):

**(S) Simple sorts:** Bubble, Insertion, Selection, Exchange
**(N) Non-comparative sorts:** Radix, Counting, Bucket
**(D) Divide-and-conquer sorts:** Merge, Quick

For group size = 1, you must implement 3 algorithms in total, consisting of 1 of your choosing from *each* of the categories S, N, and D.

For group size = 2, you must implement 6 algorithms in total, consisting of 2 of your choosing from *each* of the categories S, N, and D.

For group size = 3, you must implement 9 algorithms in total, consisting of 4 from category S, 3 from category N, and 2 from category D.

The total number of algorithms scales linearly with the size of the group, but in fact, the difficulty will decrease since several algorithms have commonalities. Your team will find it easier to develop the additional algorithms after you finish the first few.

Of course, there are *many, many* more sorting algorithms than the above. This list was selected to give you familiarity with some of the most popular ones, but we will see more throughout the class.

### Sort Program

In addition to the `SortLib` library, you will create a program, `sort`, which allows you to read a list of integers, sort them according to one of your supported algorithms, save the result, and, optionally, print the time it took to sort.

### *Report Deliverable (Canvas) - 50%*

The other half of the grade is the report. You must write a report which includes a description of the algorithms implemented in your sorting library, some theoretical analysis, experimental runtime results, and a short discussion.

## Part A - Code (Submit to Mimir)

### *Sorting Algorithms*

We will cover linear, non-comparative, and divide-and-conquer algorithms in class. There are other algorithms which we will cover later (e.g., heap sort), but many of these are associated with particular data structures that we have not yet discussed. All of the selected algorithms for this assignment can be applied to linear structures such as arrays, vectors, and linked lists.

A high-level overview of each sort is given below, but we will go into more detail in class.

#### Simple Methods

**Bubble Sort**
Compares a pair of adjacent elements and swaps right to left if it is smaller. Goes through all pairs and repeats the process *n* times to ensure every element has had the chance to move anywhere in the list.

**Insertion Sort**
Compares right to left until a smaller element is found and inserts it there. In other words, at each step, we're identifying where an element belongs in the list of elements before it.

**Selection Sort**
Finds the minimum element using linear search (recall that we can't use binary search on an unsorted list!) and moves it to the front of the unsorted part of the list. At each step, shrink the unsorted part and consider the elements moved to the front as belonging to the sorted part.

**Exchange Sort**
Compares pairs of elements, but rather than only considering adjacent elements, it searches from the current index through the entire list and swaps as smaller elements are found. Then it increments the index and compares the new element against all subsequent ones in the list.

#### Non-Comparative Methods

**Radix Sort**
Rather than compare any elements, this method creates "buckets" to hold counts of a single digit from each element. For base 10 integers, we use 10 buckets, and we place them in the bucket corresponding to their index. For example, we use an array to manage the buckets, and for the numbers 10, 15, and 12, they will go into indices 0, 5, and 2 based on their last digit, respectively. We reorder the list based on the counts in each bucket and repeat the process for each digit (10's, then 100's, etc.).

**Counting Sort**
This is the degenerative case of radix sort in which we create a bucket to hold a count for every element of the list rather than using the number base. Thus, it requires an array of size *max* - *min* at the very least. The space tradeoff enables us to only check the list and reorder it once.

**Bucket Sort**
Similar to other non-comparative methods, and namesake for the term “bucket”, this sort is the generalized case in which we create some number of buckets and reorder the list from the bucket contents. The two notable distinctions are that bucket sort stores the actual element in the bucket rather than just a count and, depending on the number of buckets, it may require a sub-sorting algorithm to keep the buckets ordered. For simplicity in this assignment, you will create the number of buckets to match counting sort with *max* - *min* elements for integers, avoiding the sub-sort issue.

### Divide-and-Conquer Methods

**Merge Sort**
As with most divide-and-conquer methods, this is a recursive algorithm. Merge sort splits the list into halves, repeatedly calling itself until the list size is 1 or 2. After this, we return up the call stack and apply a “merge” algorithm, which is just reading through the two halves and selecting the next smallest element from each in order to populate the new merged list.

**Quick Sort**
Quick sort selects a pivot value in the array and then partitions the list into sublists based on the pivot: one list for each the smaller, larger, and equal elements. It recursively calls itself, selecting new pivots and partitioning more sublists. This process repeats until there is nothing left to partition (size is 0 or 1, just the pivot). Then the sublists can be concatenated back up the call stack directly.

The pivot value is sometimes chosen as the first or last element, the midpoint, or a random number. The selection of the pivot is left to you. It is not a significant difference for a uniform, unordered list, although there are arguments for random or midpoint selections if a list is nearly-ordered.

## *Sorting Library*

In this assignment, we will also practice creating a static library that can be linked to our main program at compile time. You have previously seen managing code across headers and sources, object files when you compile from multiple sources, and Makefiles to help manage the build process. To create your library, we will combine knowledge from all of these concepts.

### SortLib.h

First, you are given the “SortLib.h” header which defines the namespace and function declarations. Any code that uses your library will know what it can or cannot call based on what is declared in the header, even though it won’t know how they are implemented until the library is provided.

You may also notice an alternative to the `#ifndef` header guards we have used previously: `#pragma once`. Though not part of the official standard, #pragma once is supported by modern compilers as an easier alternative since it is less prone to typos.

**You do not need to edit the SortLib.h file!**

### lib/*.cpp Files

The starter code includes empty source files for all of the sort functions. You only need to implement the ones required based on your group size. When building the library, it actually won't matter if not all definitions are provided since only the ones you call will be searched for.

To implement a sort function, follow these steps:

- Include "SortLib.h" at the top to add the namespace and function declaration
- Copy the templated function declaration over, ensuring that it is scoped to the `SortLib::` namespace, as you would with a class or any other function definition outside of scope.
- Add a function instantiation for "`int`" data types at the bottom of the source file so that the library compiles with an actual definition
    - Similar to our previous assignment where you added the template class instantiation at the bottom of the file, you'll need one for the function. If you leave this out, you will actually receive a warning when packing the library that it has no definitions! Again, templates are just a pattern. The compiler won't generate a pattern for a given type unless you tell it to.
    - Defining a specific instance of a function template is even easier; you just repeat the name with "template" in front. For example, for BubbleSort, it would look like this: `template void SortLib::BubbleSort(int*, int);`

After those steps, you can now write your sort algorithm!

Notice that all sort algorithms are defined with the same parameters. They all expect a templated array and its size. These algorithms can be implemented with dynamic arrays, and most of them can operate on the array in-place.

**For simplicity, in this assignment, you need only instantiate "int" data types in your library, so it is ok for you to only create the non-comparative algorithms according to integer types.**

## *Makefile*

The next step in preparing your library is to fill the Makefile.

### Archive Utility - ar

You will want a command that compiles the object files for all your sort algorithms. An easy way to do this is with the Unix wildcard, which is an asterisk (*): `g++ -std=c++17 -c *Sort.cpp.` The *Sort.cpp matches all the source files ending in Sort.cpp, and the -c creates the object files (.o files) for each.

Next, we will combine all the object files into a single library package. To do this, we use the archiver[1] Unix utility. The command line utility for archiver is simply "ar", and we will use the following command to pack the files into a static library called `SortLib.a`:

```
ar rvs SortLib.a *Sort.o
```

[1] https://en.wikipedia.org/wiki/Ar_(Unix)

This tells `ar` to insert all the .o files into the archive with replacement (r), writing the file whether it changes or not (s), using verbose mode to print out as much information as possible (v). Verbose mode is what enables helpful warnings, like if you're missing instantiations of a templated type.

## Static and Dynamic Libraries

### Including Headers

When you're dealing with libraries in C++, you must always include the header files. You have been doing this from the beginning with system files (which use the angled brackets <> to search for header files provided by the system), and later you started doing it with your own header files (using double quotes "" to find .h files in your local directory tree).

Recall that #include is simply a preprocessor macro that tells the compiler to copy the contents of the file into that part of the source file. Headers enable your source code to know exactly what classes, functions, etc. can be used and what parameters they'll accept and return.

### Including Other Sources

When it comes to providing the function definitions, we have previously been compiling the sources for our own headers and linking them with the main program in the same step. This is what is happening when you pass all the source files to your g++ command at the same time.

You could always distribute your library source code, as many open source projects do, so that anyone wanting to use your code could compile the library code directly into their source. There is a possible advantage to this when it comes to working with templates: putting everything in .h files that you distribute can side-step the problem of supporting uninstantiated data types. The headers would only be pulled into code that uses them, so the compiler would generate instantiations based on that.

However, this is not the best idea. First of all, it defies the preferred guidance of separating declarations and definitions into header and source files. Secondly, even if you use header and source, you don't always want to distribute your source code. Many companies sell pre-compiled software programs as products, and some companies provide rich libraries and APIs (application programming interfaces) as products. They usually do not want to distribute their source code. Pre-compiled libraries are the solution.

### Distributing Libraries

Libraries can be either *static* or *dynamic*. A static library typically ends in ".a" and is needed at compile time. This is very similar to providing the source code that is compiled into your program but you use a binary object file that the compiler can link. It is akin to giving someone a compiled program: technically, everything is there, but to reverse engineer the source is nearly impossible (although hacking and disassembling could give you some idea).

A dynamic library is one that is loaded at runtime. Dynamic libraries are typically seen as ".so" in Linux, ".dll" in Windows, or ".dylib" in Mac (with some minor differences). A major advantage of dynamic libraries is that the library code can be updated independently of the main program. For instance, you can update a .dll file in Windows without changing the program that calls it. (Speaking of hackers and disassemblers, shared libraries are one potential method of changing source program behavior since you could create a shared library that presents the same interface but functions slightly differently.)

This short essay on libraries is mostly informational. I congratulate any of you who read all of it!

## Makefile Contents

You are given the directory structure and SortLib header in the starter code. The Makefile starts as blank, but you can add the commands we discussed as targets. You'll want a target to make the library, one to make the main program, and one to clean the files generated by the build process.

For convenience, you may use the following as your Makefile:

```
CC=g++
CFLAGS=-std=c++17 -g
LIBPATH=./lib

all: main

main: lib sort
	$(CC) $(CFLAGS) -o sort sort.o -L$(LIBPATH) $(LIBPATH)/SortLib.a

lib: SortLib.a

SortLib.a: lib/*Sort.cpp
	cd $(LIBPATH) && $(CC) $(CFLAGS) -c *Sort.cpp && \
		ar rvs SortLib.a *Sort.o

sort: sort.cpp
	$(CC) $(CFLAGS) -I$(LIBPATH) -c sort.cpp

clean:
	rm -f sort sort.o $(LIBPATH)/SortLib.a $(LIBPATH)/*.o
```

This defines several variables for ease, such as the library path. If you look at makefiles for large open source projects, you'll see how powerful and complex they can become through the use of variables and targets that call other targets.

A couple things to note:

- The && between commands in the SortLib.a target join the commands to run consecutively and require each one to execute successfully before running the next. For example, you must change into the library directory to build the object files for your sort algorithms. If these are on separate lines without the &&, the cd into lib will not apply to the compile command.
- The sort target introduces a critical g++ argument when working with libraries: `-I`. The `-I` command extends the search path for included headers to the provided directories. In this case, it resolves to `-I./lib`, which is where our SortLib.h header is. Note that there is also a corresponding -L that we use in the main target when linking. This extends the library search path so we can pass SortLib.a to the linking step.

### *Sort Program*

#### sort.cpp

The final portion of the code activity is to write the `sort` program. This program compliments your distributable SortLib.a library, and it demonstrates using SortLib to sort and time different algorithms on a multitude of inputs.

The sort.cpp file only includes a few lines of code, but there are many helpful hints in the comments. You don't have to follow the comments. They are optional scaffolding. The main requirement is that you meet the input/output criteria.

The key functionality of `sort` is defined below:

- Write a program, whose binary executable is named "sort", capable of accepting the following arguments:
    - **-a to specify a sorting algorithm**
        - bubble for BubbleSort, insertion for InsertionSort, merge for MergeSort, etc.
        - If an unrecognized algorithm is given, print the following to cerr and return 1:
          `Unsupported sorting algorithm.`
    - -i to specify an optional input file
    - -o to specify an optional output file
    - -t to optionally specify if timing information should be printed
- If -a and the sorting algorithm are not provided, print out the following on cerr and return 1:
  `Usage: sort -a <algorithm> [-i <infile> -o <outfile> -t]`
- If -i and a filename are not provided, you will read from stdin (cin); otherwise, you must open and read from the file.
    - **You must support an input of integer numbers**. The first number you read will be the size of the list. The remaining numbers will be the list contents that must be sorted.
    - You can assume that if -i is present, there will also be a filename. If a file fails to open, print out the following on cerr and return 1:
      `Unable to open input file.`
- If -o and a filename are not provided, you will write to stdout (cout); otherwise, you must open and write to the file.
    - **The output will print the sorted list**, without the leading number for size. You will just print each number, in sorted order, followed by a space. The last number should be followed by an endline rather than a space.
    - You can assume that if -o is present, there will also be a filename. If a file fails to open, print out the following on cerr and return 1:
      `Unable to open output file.`
- If -t is provided, print the elapsed time in microseconds as "Elapsed Time: <duration>" at the very end of the program.
    - **It is recommended to use the `chrono` library for tracking runtime.**

- Included in C++11, chrono lets you save the time at key points in your program with something like the following code:
  `auto start = std::chrono::high_resolution_clock::now();`
- Save the time right before you call your sort algorithm and again right after it returns so you can calculate the total duration to be displayed.
- Assuming your start time was saved in “start” and stop time in “stop”, you could do something like the following to get the duration in microseconds.
  ```
  auto duration =
  std::chrono::duration_cast<std::chrono::microseconds>(stop -
  start);
  ```
- We are using the “auto” keyword in this case to let the compiler select the data type most appropriate for your system, rather than potentially casting the time into a non-optimal data type.

### *Point Distribution*

This part comprises 50% of the overall assignment score. That is distributed as follows:

- 30% - Autograded
  - 5% for the simple algorithm(s)
  - 5% for the non-comparative algorithm(s)
  - 5% for the divide-and-conquer algorithm(s)
  - 15% for the sort program
- 20% - Manual Code Review
  - 5% for the simple algorithm(s)
  - 5% for the non-comparative algorithm(s)
  - 5% for the divide-and-conquer algorithm(s)
  - 5% for the sort program
  - Even though only 5% is set aside for manual review of each portion, note that autograded points can be revoked if attempts to circumvent the spirit of the assignment’s autograding are discovered.
  - These should be easy points just by commenting code, making things readable, and following the assignment goals.

## Part B - Report (Submit to Canvas)

You and your team should write a report containing the following sections.

### *Team Information and Sources*

List all team members and any sources you consulted: other students, website links, books, or any other reference materials.

### *Introduction*

Briefly describe the assignment objective, your code structure, and the algorithms you implemented. For each algorithm, provide a short description of how it works.

### *Theoretical Analysis*

In this section, provide an analysis of the complexity of each sorting algorithm you implemented for different input types. This does not have to be a rigorous mathematical proof, but it should use clear reasoning based on the structure and behavior of the algorithm if not providing a mathematical basis. For the input types, consider the conditions of the best, worst, and average case inputs and describe what inputs result in these different cases.

You should provide a table with the O() complexity of each of your sort algorithms for best, worst, and average cases. I recommend your table use the following as a basis:

| **Complexity** | Best | Average | Worst |
|---|---|---|---|
| **Bubble Sort** | | | |
| **…** | | | |

### *Experimental Results*

This is the most critical section of the report. For each of the algorithms you have implemented, you should compare the performance (runtime) of the sorting operations on different input lists.

First, give the specifications of your test machine. Runtime results will vary depending on the machine running the test, so you should use the *same* machine for all timing information.

Second, run each of your sorting algorithms against the provided test 0-4 input files **3** times. These include random lists of 100, 1,000, 10,000, 20,000, and 100,000 integers, respectively. Note that you can generate your own lists of integers in C++ using the rand() function, bounded by mod to get a number within a set range, e.g., repeat this operation 100 times, rand() % 1000, to get 100 numbers between 0 and 1000.

Third, create a table which shows the runtime for each of the 3 trials for each of the algorithms you implemented on each of the input files. I recommend your table use the following as its basis:

| **Algorithm** | $10^2$ | | | $10^3$ | | | $10^4$ | | | $2x10^4$ | | | $10^5$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 | Trial 1 | Trial 2 | Trial 3 |
| **Bubble** | | | | | | | | | | | | | | | |
| **…** | | | | | | | | | | | | | | | |

Fourth, plot the average runtime for each of your sorting algorithms relative the size of the array. Runtime is y-axis and size of the array is x-axis. You must create a computerized plot, not a hand-written one.

Provide both linear and logarithmic plots.

### *Discussion and Conclusion*

Provide a discussion of your results.

- How did your experimental results compare with your theoretical analysis? Are there any discrepancies?
- Which sorting algorithm performs best for which case? Does it depend on the inputs? Why or why not?
- What factors could affect your experimental results?

You can include any other information you think is relevant.

### *Point Distribution*

This part comprises 50% of the overall score and will be graded manually as follows:

- 5% - Team information and sources
- 5% - Introduction and algorithmic descriptions
- 10% - Theoretical analysis
- 25% - Experimental results
    - 15% - Runtime analysis table
    - 10% - Linear and logarithmic plots
- 5% - Discussion and conclusion

## Quick Start Guide

1. Select and set up your team.
    a. Guide: Create your group in Canvas
    b. Guide: Create your group in Mimir
    c. Suggestion: Use github.tamu.edu to organize your team's codebase. Use a private repository available to only your team.
2. Go to the assignment on Mimir and download the starter code.
3. Select the sorting algorithms you (and your teammates) will be implementing.
    a. You can separate out the tasks or work collaboratively through them. You will find that having one algorithm in a category written will make it easier to write others of that same type.
    b. You should reference this document in more detail for implementing your algorithms and filling in the Makefile. It can save some time.
4. Implement the sort program as specified in this prompt.
5. Submit your code to Mimir
    a. Update the GROUP_SIZE in Group.txt for the number of members in your group.
    b. **Note**: You may need to submit your Mimir code as a zip file if it does not support uploading your lib subdirectory alongside the other files.
6. Write the report.
    a. Suggestion: Use Google Docs to enable collaborative writing on the report.
    b. Select who will be running the timing analysis and run all tests on the same machine.
    c. Fill in the tables and plots alongside the required discussion.
7. Submit your report to Canvas

Regarding automated grading, it's looking like Mimir won't provide the form of grading needed to support groups and choice for algorithms. It might be possible to create a custom solution, but we will take the easier path of providing a diagnostic test for all algorithms. These will be worth 0 points but will provide the grader with the equivalent of autograding to assign the points in the manual review.