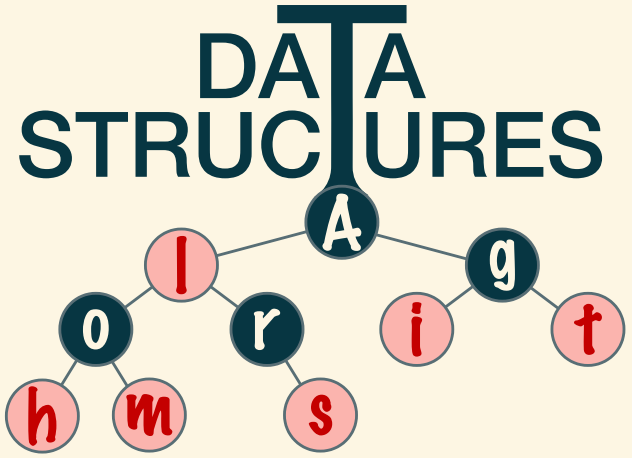
Home



WEEK 01: FOUNDATIONS (LAB)

Michael T. Gastner

2026-01-08

Home

INTRO

Home

LEARNING OBJECTIVES

By the end of this lab, you should be able to:

1. Understand and implement merge sort, including its pseudocode and C++ implementation.
2. Submit C++ programs to Gradescope and analyze the autograder's feedback.
3. Develop a recursive implementation of insertion sort, reinforcing concepts of recursion in algorithm design.
4. Compare the running times of algorithms.

Home

OVERVIEW OF LAB ACTIVITIES

Upload solutions for the following activities to xSITe or Gradescope:

1. Merge Sort—Fill in the Gaps
   Complete a partially pre-filled diagram of the merge sort algorithm.
2. Testing the Gradescope C++ Autograder
   a. Upload and test the provided merge sort code.
   b. Write and upload a basic "Hello, World!" program.
3. Recursive Implementation of Insertion Sort
   a. Write the Pseudocode.
   b. Implement it as a C++ program.
4. Comparing Running Times
   Run a provided C++ program and interpret a plot of running time versus input size.

None of today's activities will be graded, but they will serve as practice for future lab activities. Upload solutions to familiarize yourself with the submission process.

Home

RECAP: MERGE SORT

Home

RECAP: MERGE SORT

URL: https://www.youtube.com/watch?v=NwH1d1YWHtE

Home

PSEUDOCODE FOR MERGE SORT

Note that the video and our textbook Cormen *et al.* (2022) adopt 1-based indexing. Here is the pseudocode for merge sort for 0-based indexing, as used by https://apps.michael-gastner.com/merge-sort/:

```
  Input:  a, array of keys
  Output: a, sorted in ascending order

1 n ← length(a)
2 sort(a, 0, n − 1)

  procedure sort(a, l, r)
    // l and r are the left and right boundary indices
    // respectively; m is the midpoint
    // Postcondition: a[l..r] sorted
3   if l ≥ r
4     return
5   m ← floor((l + r) / 2)
6   sort(a, l, m)
7   sort(a, m + 1, r)
8   merge(a, l, m, r)
```

Home

MERGE SORT: A DIVIDE-AND-CONQUER ALGORITHM

Merge sort is a **divide-and-conquer algorithm** because it breaks a problem into smaller, related subproblems and then combines these solutions to solve the original problem.

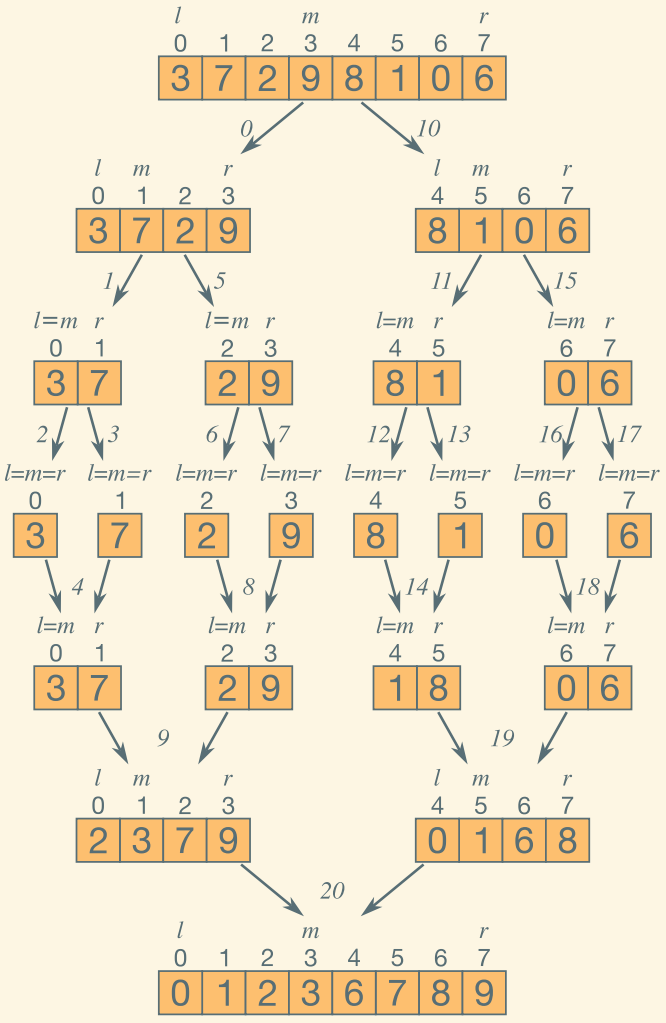
Divide-and-conquer algorithms are **recursive**. For instance, the `sort` procedure in the pseudocode on the previous slide calls itself in lines 6 and 7.

Home

EXAMPLE: MERGE SORT APPLIED TO AN ARRAY

The example on the right illustrates the merge sort algorithm applied to a numeric array, modeled after Figure 2.4 in Cormen *et al.* (2022).

Italicized numbers indicate the sequence in which the `sort()` and `merge()` procedures are invoked, starting with the initial call to `sort(a, 0, 7)`.



Home

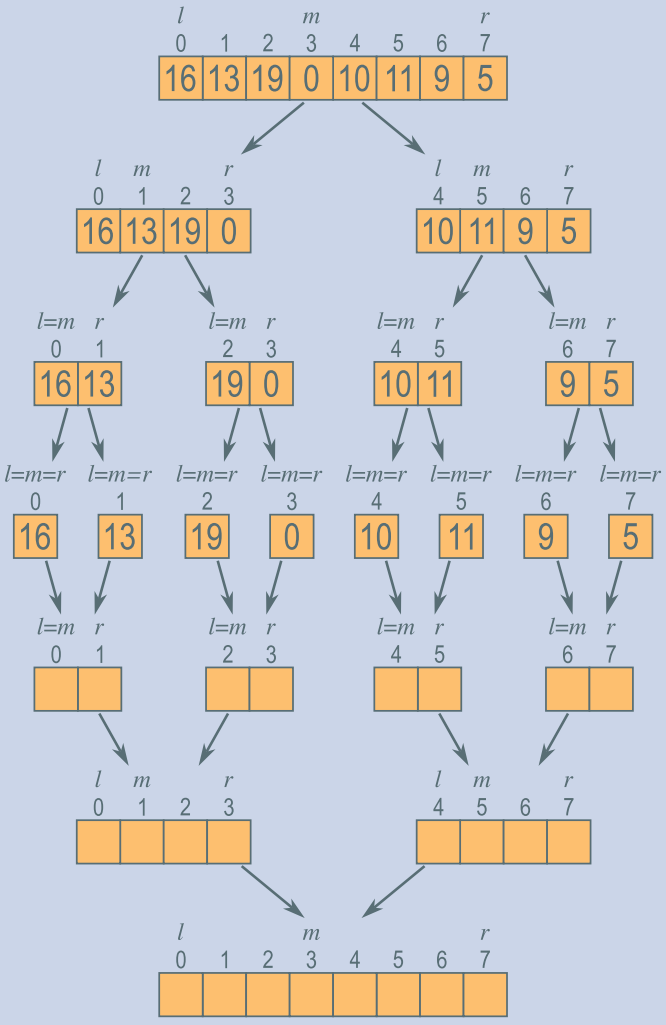
EXERCISE 1: MERGE SORT—FILL IN THE GAPS

Download the image on the right from GitHub Pages in PNG or PDF format. Adopting the pattern from the example on the previous page, complete the following tasks:

a. Label the arrows with numbers indicating the sequence of the `sort()` and `merge()` procedure calls.

b. Fill in the correct values in the empty boxes.

You may use any drawing tool, such as Annotely. When finished, upload the completed image (e.g., as a screenshot) to xSITe (Dropbox → Week 01 Lab Exercise 1: Merge Sort—Fill in the Gaps).



MERGE SORT IN C++

Home

C++ IMPLEMENTATION OVERVIEW

As with the insertion sort implementation discussed in the lecture, we split the merge sort implementation into multiple files to enhance code readability and maintainability. You can download the complete code as a ZIP archive.

`main.cpp`
: Entry point of the program

`import_numbers_from_file.cpp` and `import_numbers_from_file.h`
: Functions to import numbers from a file, one number per line

`merge_sort.cpp` and `merge_sort.h`
: Implementation of the `merge()` and `sort()` procedures

`makefile`
: File to compile the program using `make`

Home

MERGE SORT'S `main.cpp` (PART 1)

The first part of `main.cpp` includes the headers, defines exit codes, and validates command-line arguments:

```cpp
#include <cstdlib>
#include <exception>
#include <iostream>
#include <string>
#include <vector>

#include "import_numbers_from_file.h"
#include "merge_sort.h"

int main(int argc, char *argv[]) {
  enum ExitCode {
    ok = EXIT_SUCCESS,
    fail = 1,
    usage = 2,
    io = 3,
    bad_input = 4,
  };

  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " <filename>\n";
    return usage;
  }
```

Home

MERGE SORT'S `main.cpp` (PART 2)

The second part reads numbers from a file, sorts them using merge sort, and prints the result:

```
24    try {
25      const std::string filename = argv[1];
26      std::vector<int> a = importNumbersFromFile(filename);
27      if (!a.empty()) {
28        mergeSort(a, 0, a.size() - 1);
29      }
30
31      for (std::size_t k = 0; k < a.size(); ++k) {
32        if (k > 0) {
33          std::cout << ", ";
34        }
35        std::cout << a[k];
36      }
37      std::cout << '\n';
38      return ok;
39    } catch (const std::runtime_error &ex) {
40      std::cerr << "Error: " << ex.what() << "\n";
41      return io;
42    } catch (const std::exception &ex) {
43      std::cerr << "Error: " << ex.what() << "\n";
44      return bad_input;
```

Home

merge_sort.cpp: THE merge() FUNCTION (PART 1)

The merge() function creates temporary arrays and copies elements:

```cpp
#include "merge_sort.h"

#include <cstddef>
#include <vector>

static void merge(std::vector<int> &a, std::size_t l, std::size_t m,
                  std::size_t r) {
  const std::size_t n_left = m - l + 1;
  const std::size_t n_right = r - m;
  std::vector<int> left(n_left);
  std::vector<int> right(n_right);
  for (std::size_t j = 0; j < n_left; ++j) {
    left[j] = a[l + j];
  }
  for (std::size_t k = 0; k < n_right; ++k) {
    right[k] = a[m + 1 + k];
  }
  std::size_t i = l;
  std::size_t j = 0;
  std::size_t k = 0;
```

Home

merge_sort.cpp: THE merge() FUNCTION (PART 2)

The function then merges the sorted subarrays back into the original array:

```
21    while (j < n_left && k < n_right) {
22      if (left[j] <= right[k]) {
23        a[i] = left[j];
24        ++j;
25      } else {
26        a[i] = right[k];
27        ++k;
28      }
29      ++i;
30    }
31    while (j < n_left) {
32      a[i] = left[j];
33      ++i;
34      ++j;
35    }
36    while (k < n_right) {
37      a[i] = right[k];
38      ++i;
39      ++k;
40    }
41  }
```

Home

merge_sort.cpp: THE mergeSort() FUNCTION

The `mergeSort()` function recursively divides the array and calls `merge()`:

```cpp
void mergeSort(std::vector<int> &a, std::size_t l, std::size_t r) {
  if (l >= r) {
    return;
  }
  const std::size_t m = l + (r - l) / 2;
  mergeSort(a, l, m);
  mergeSort(a, m + 1, r);
  merge(a, l, m, r);
}
```

Home

GRADESCOPE

Home

EXERCISE 2A: TESTING GRADESCOPE (MERGE SORT)

To prepare for future assignments, navigate to the Dropbox folder “Week 01 Lab Exercise 2a: Gradescope Demo—Merge Sort” on xSITe. Download the files as instructed in the assignment. Note that this is a practice assignment and will not be graded.

For simplicity, the solution file `merge_sort.cpp` is provided in the ZIP archive you downloaded earlier.

Upload the sample solution to Gradescope and review the autograder’s feedback. You can resubmit code as often as needed to familiarize yourself with the submission process. You may want to intentionally introduce an error to observe how the autograder responds.

Home

EXERCISE 2B: TESTING GRADESCOPE (“HELLO, WORLD!”)

Write a C++ program, consisting of a single file titled `hello_world.cpp` that prints “Hello, World!”—exactly these 13 characters, followed by a newline character—to the `stdout` console. Compile the code using `makefile` provided in the ZIP archive.

A template for `hello_world.cpp` is also provided in the ZIP archive. This template won’t print anything. Your task is to update the file.

Follow the instructions in the Dropbox folder “Week 01 Lab Exercise 2b: Gradescope Demo—Hello, World!” on xSITe to prepare your submission. Again, this is a practice assignment and will not be graded.

Home

RECURSIVE INSERTION SORT

Home

EXERCISE 3A: RECURSIVE INSERTION SORT—PSEUDOCODE

The lecture introduced insertion sort as the non-recursive algorithm shown below.

However, insertion sort can also be implemented as a recursive algorithm. To sort `a[0..n]`, recursively sort the subarray `a[0..(n-1)]` and then insert `a[n]` into the sorted subarray. Write pseudocode for this recursive version. Submit your solution (e.g., a photo of your handwritten notes) to xSITe (Dropbox → Pseudocode of Recursive Insertion Sort).

```
  Input:  a, array of keys
  Output: a, sorted in ascending order

1 n_keys ← length(a)

  // Loop invariant (before each iteration): a[0 .. i - 1] is sorted
2 for i ← 1 .. n_keys - 1
3   active_key ← a[i]

    // Insert active_key into sorted subarray
4   h ← i  // h is the hole index
5   while h > 0 and a[h - 1] > active_key

      // Fill hole: shift left neighbor to the right
6     a[h] ← a[h - 1]
7     h ← h - 1
8   a[h] ← active_key
```

Home

EXERCISE 3B: RECURSIVE INSERTION SORT IN C++

Implement insertion sort in C++ as a recursive function. Your implementation should be contained in a file named `recursive_insertion_sort.cpp`. A template file, together with other utilities for compilation are provided in this ZIP archive.

Upload your solution to Gradescope. Detailed instructions are provided on xSITe (Dropbox → Week 01 Lab Exercise 3b: Gradescope—Recursive Insertion Sort in C++).

Home

COMPARING RUNNING TIMES

Home

COMPARING RUNNING TIMES

We have studied three implementations for sorting an array of numbers:

- Non-recursive insertion sort
- Recursive insertion sort
- Merge sort

Which algorithm is best in practice? The choice depends on the size of the input array and the relative efficiency of the algorithms, particularly in terms of running time and memory usage.

To evaluate their performance, we will compare the running times of the three algorithms for sorting arrays of various sizes. Using the `std::chrono` library, we can measure these running times with a high-resolution clock for precise time measurement.

Home

C++ CODE FOR COMPARING RUNNING TIMES

Code for running-time measurements is available as a ZIP archive, containing the following files:

- Entry point of the program: `main.cpp`
- Sorting algorithms:
  - `insertion_sort.cpp` and `insertion_sort.h`
  - `recursive_insertion_sort.cpp` and `recursive_insertion_sort.h`
  - `merge_sort.cpp` and `merge_sort.h`
- Utilities for plotting: `vega_lite_plot.cpp`, `vega_lite_plot.h`, and `nlohmann_json.h`
- File to compile the project: `makefile`

Home

EXPLANATION OF THE C++ PROGRAM

When you execute `make run`, the program generates arrays of random numbers and sorts them using non-recursive insertion sort, recursive insertion sort, and merge sort. The running times for arrays of different sizes are measured and stored in a JSON file. You can visualize the data by opening the generated file `vega_lite_plot.html` in a web browser. A sample run is shown on the following page.

Home

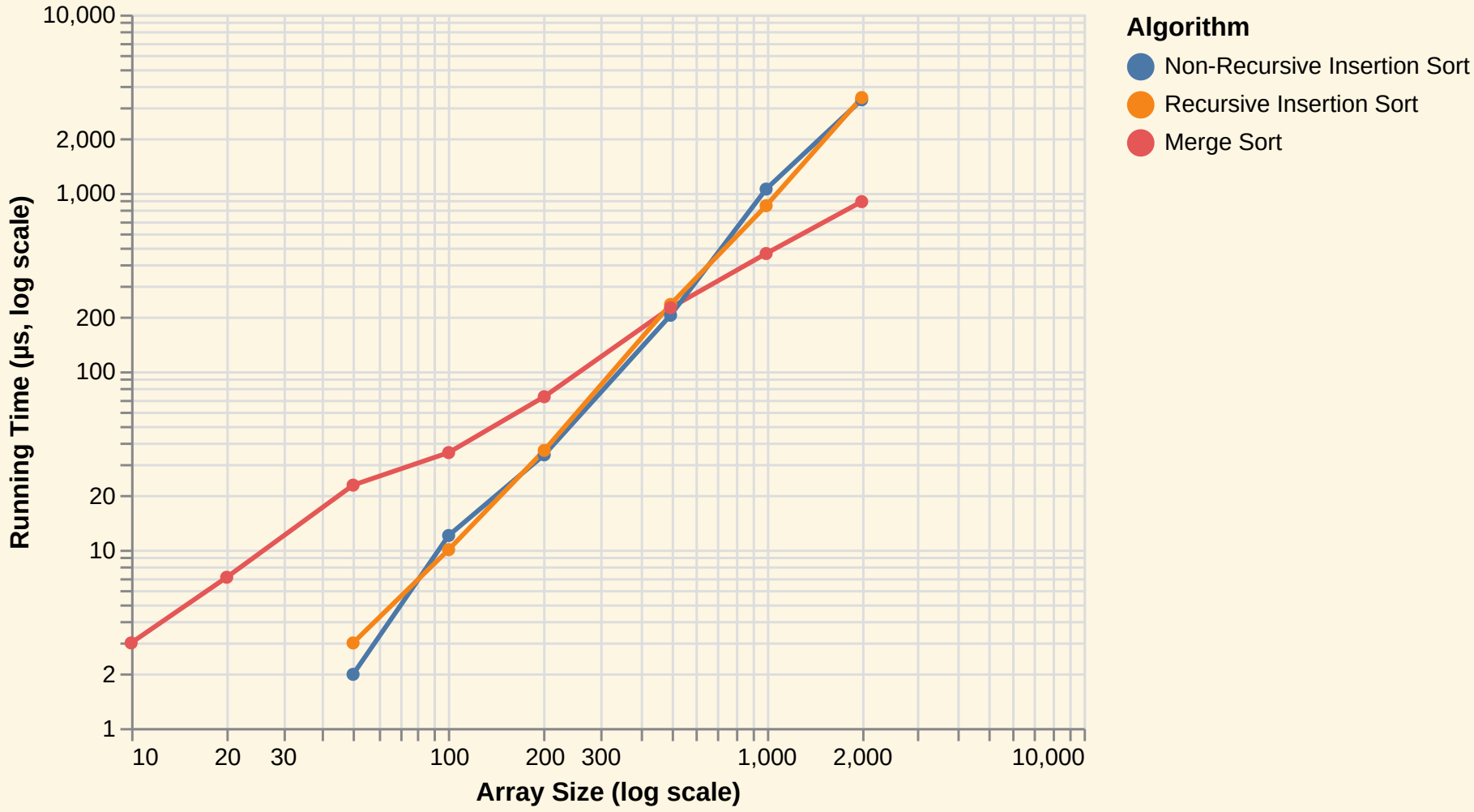
SAMPLE RUN OF THE C++ PROGRAM

Array Size	Insertion Sort (µs)	Recursive Insertion Sort (µs)	Merge Sort (µs)
10	0	0	3
20	0	0	7
50	2	3	23
100	12	10	35
200	34	36	72
500	206	237	227
1000	1050	847	457
2000	3325	3421	893

On my computer, both versions of insertion sort exhibit approximately equal running times. Merge sort is slower on small arrays but faster on larger arrays. The representative plot on the following slide confirms this observation.

Home

PLOT OF RUNNING TIME VERSUS ARRAY SIZE



Home

CONCLUSION

Home

SUMMARY OF KEY LEARNING OUTCOMES

1. Reviewed the merge sort algorithm and its pseudocode. ↪
2. Explored the multi-file C++ implementation of merge sort. ↪
3. Submitted C++ programs to Gradescope and interpreted the autograder's feedback (merge sort and "Hello, World!"). ↪
4. Developed a recursive insertion sort (pseudocode and C++). ↪
5. Compared the running times of non-recursive insertion sort, recursive insertion sort, and merge sort. ↪

Home

OUTLOOK

Next week, you will learn to analyze the running times of algorithms and express them using mathematical notation. The lecture and lab will cover sections of Chapters 3 and 4 in Cormen *et al*. (2022).

BIBLIOGRAPHY

- Cormen, T.H. *et al*. (2022) *Introduction to algorithms*. 4th ed. MIT Press.