Backend Engine & Logging System Guide

Introduction

This document explains how the Interactive Sorting Algorithm Visualizer's backend works. You'll learn how the C++ engine executes sorting algorithms, captures every step for visualization, and exports comprehensive JSON logs for the frontend to animate.

---

1. Overall Backend Architecture

The Big Picture

The backend is a C++ console application (`sortgen.exe`) that functions as a "sorting laboratory." When executed, it:

1. Reads command-line arguments (algorithm name, input file, output file)
2. Loads an array of integers from a dataset file
3. Executes the requested sorting algorithm
4. Captures every comparison, swap, and array state change
5. Writes a detailed JSON log file for frontend visualization

Think of it as a recording studio for sorting algorithms—the algorithms perform their work while every action is meticulously documented.

Component Roles

main.cpp — The Orchestrator This is the entry point. It handles:

- Parsing command-line arguments
- File I/O (reading input data, writing output logs)
- Creating and coordinating the StepRecorder and Metrics objects
- Delegating to SortEngine to run the chosen algorithm
- Timing the execution
- Packaging metadata (start time, total comparisons, complexity info)

SortEngine.cpp — The Dispatcher This acts as a factory or dispatcher. It receives an algorithm name as a string (like "bubble" or "merge") and routes execution to the appropriate algorithm function. This design pattern

allows easy extensibility—adding a new algorithm just means adding a new function and one more branch in the dispatcher.

```cpp
void SortEngine::runAlgorithmByName(const string& name, vector<int>& arr,
                                    StepRecorder& rec, Metrics& metrics) {
    if (name == "bubble") {
        bubbleSort(arr, rec, metrics);
    } else if (name == "merge") {
        mergeSort(arr, rec, metrics);
    }
    // ... more algorithms
}
```

algorithms/*.cpp — The Algorithm Implementations Each file contains one sorting algorithm. The algorithms are implemented as static methods of the `SortEngine` class. All algorithms share the same signature:

```cpp
static void algorithmName(vector<int>& arr, StepRecorder& rec, Metrics& metrics);
```

This uniform interface makes them interchangeable and easy to manage.

Runtime Selection Mechanism

The project uses runtime polymorphism through string-based dispatch rather than function pointers or virtual functions. This is intentional for educational clarity. When a user runs:

```bash
sortgen.exe bubble input.txt output.json
```

The flow is:

1. main.cpp parses "bubble" from argv[1]
2. Calls `SortEngine::runAlgorithmByName("bubble", ...)`
3. SortEngine's if-else chain matches "bubble" and calls `bubbleSort(...)`
4. The algorithm executes while recording steps
5. Results are flushed to output.json

---

2. Step Recording System

What is a "Step" or "Snapshot"?

A **Step** represents a single moment in time during the sorting process. It's a photograph of:

- The current state of the entire array
- Which elements are being compared (if any)
- Which elements are being swapped (if any)
- What type of operation is occurring
- Cumulative metrics (total comparisons and swaps so far)
- Special markers (like pivot indices for Quick Sort)

The Step structure is defined in `Step.hpp`:

```cpp
struct Step {
    vector<int> array;              // Full array state
    vector<int> compare_indices;    // Elements being compared
    vector<int> swap_indices;       // Elements being swapped
    string operation;               // "compare", "swap", "overwrite", etc.
    size_t cumulative_comparisons;
    size_t cumulative_swaps;
    optional<int> pivot_index;      // For Quick Sort
    vector<int> sorted_indices;     // Elements in final position
};
```

Why Record Every Step?

This project is designed for **educational visualization**, not production use. The goal is to show students exactly how sorting algorithms work at a granular level. Therefore:

- Every comparison is recorded before it happens
- Every swap is recorded after it happens
- Every array modification is captured
- Initial and final states are always included

In a production system, this would be wasteful. But for learning, seeing every single operation is invaluable—students can step through the algorithm frame-by-frame like watching a movie with frame-by-frame control.

How Logging Works

The `StepRecorder` class (in `step_recorder/StepRecorder.cpp`) maintains a vector of Step objects:

```cpp
class StepRecorder {
private:
    vector<Step> steps;

public:
    void record(const Step& s) {
        steps.push_back(s);
    }
};
```

Each algorithm calls `rec.record()` at strategic points. For example, in Bubble Sort:

```cpp
// Before comparison
metrics.comparisons++;
rec.record({arr, {j, j+1}, {}, "compare", metrics.comparisons, metrics.swaps});

// After swap
if (arr[j] > arr[j+1]) {
    swap(arr[j], arr[j+1]);
    metrics.swaps++;
    rec.record({arr, {j, j+1}, {j, j+1}, "swap", metrics.comparisons, metrics.swaps});
}
```

Notice that `rec.record()` receives an inline-initialized Step struct using aggregate initialization. This is a clean C++ idiom for constructing objects without explicit constructors.

The Metrics Structure

Alongside the StepRecorder, a `Metrics` struct tracks cumulative counts:

```cpp
struct Metrics {
    size_t comparisons = 0;
    size_t swaps = 0;
};
```

Algorithms increment these counters and pass the current values when recording steps. This allows the frontend to display real-time statistics.

Operation Types

Different operations have different visual meanings:

- **"none"**: Initial state or final state, no highlighting
- **"compare"**: Elements being compared (highlighted in one color)
- **"swap"**: Elements being swapped (highlighted and animated swapping)
- **"overwrite"**: Single element being written (used in Insertion, Merge)
- **"pivot"**: Current pivot element in Quick Sort
- **"to_bucket"**: Element being distributed in Bucket Sort
- **"mark_sorted"**: Element confirmed in final position

The frontend interprets these operations to apply appropriate colors and animations.

---

3. JSON Logging

Why JSON?

JSON (JavaScript Object Notation) is a text-based, human-readable format that's:

- Easy to parse in JavaScript (native for web frontends)
- Human-readable for debugging
- Structured yet flexible
- Language-agnostic (C++ can write it, JavaScript can read it)

The project uses the `nlohmann/json` library, which provides intuitive C++ syntax for JSON manipulation.

JSON Structure

Every output file contains two main sections:

1. Metadata — Algorithm information and summary statistics

```
json
```

```
{
 "metadata": {
  "algorithm": "bubble",
  "input_size": 10,
  "seed": 12474265,
  "start_time": "2025-12-09T22:00:00Z",
  "total_time_ms": 0,
  "total_comparisons": 42,
  "total_swaps": 15,
  "time_complexity": "O(N^2)",
  "space_complexity": "O(1)"
 }
}
```


2. Frames — Array of all captured steps


```
json

{
 "frames": [
  {
   "array": [17, 63, 63, 80, 28, 10, 83, 81, 17, 99],
   "operation": "compare",
   "cumulative_comparisons": 1,
   "cumulative_swaps": 0,
   "highlights": {
    "compare": [0, 1]
   }
  },
  // ... hundreds or thousands more frames
 ]
}
```


What Each Frame Contains

- **array**: Full array state at this moment
- **operation**: What's happening ("compare", "swap", etc.)
- **cumulative_comparisons**: Total comparisons made so far
- **cumulative_swaps**: Total swaps made so far
- **highlights**: Object containing arrays of indices to highlight

- `compare`: Indices being compared
- `swap`: Indices being swapped
- `pivot`: Single pivot index (for Quick Sort)

- **sorted_indices** (optional): Indices known to be in final position

How JSON is Generated

The `StepRecorder::flushToFile()` method serializes everything:

```cpp
```

```
void StepRecorder::flushToFile(const string& path, const Metadata& m) {
    json j;

    // Metadata section
    j["metadata"] = {
        {"algorithm", m.algorithm},
        {"input_size", m.input_size},
        // ... more fields
    };

    // Frames section
    j["frames"] = json::array();
    for (const auto& step : steps) {
        json frame;
        frame["array"] = step.array;
        frame["operation"] = step.operation;
        frame["cumulative_comparisons"] = step.cumulative_comparisons;
        frame["cumulative_swaps"] = step.cumulative_swaps;

        json highlights;
        if (!step.compare_indices.empty())
            highlights["compare"] = step.compare_indices;
        if (!step.swap_indices.empty())
            highlights["swap"] = step.swap_indices;
        if (step.pivot_index.has_value())
            highlights["pivot"] = step.pivot_index.value();

        frame["highlights"] = highlights;
        j["frames"].push_back(frame);
    }

    ofstream file(path);
    file << j.dump(2); // Pretty print with 2-space indentation
}
```

The nlohmann/json library automatically handles type conversions and JSON syntax, making the code remarkably clean.

Why JSON Files Are Large

For a 100-element array sorted with Bubble Sort, you might have 5,000+ frames. Each frame contains:

- The full 100-element array (not just changes)

- Metadata about the operation
- Index information

This is intentional. The tradeoff is:

- **Advantage**: Frontend implementation is simple—just load and display frames
- **Disadvantage**: Large file sizes (can be several MB for 100+ elements)

For this educational tool, simplicity and completeness trump file size optimization.

How Frontend Consumes Logs

The frontend (not covered in depth here) simply:

1. Fetches the JSON file
2. Parses it with `JSON.parse()`
3. Displays array elements as visual bars
4. Steps through frames with timing controls
5. Applies colors based on the `operation` and `highlights` fields

The backend's comprehensive logging makes frontend development straightforward.

---

4. File Responsibilities

algorithms/bubble.cpp, selection.cpp, insertion.cpp

These implement the three basic $O(n^2)$ comparison sorts. They follow nearly identical patterns:

- Outer loop controls passes
- Inner loop performs comparisons
- Record steps before comparisons and after swaps
- Use "compare" and "swap" operations

Key insight: These algorithms are simple enough that the instrumentation is clear. Students can see where `rec.record()` calls mirror the algorithm's logic.

algorithms/merge.cpp

Implements divide-and-conquer Merge Sort. Key features:

- Uses auxiliary arrays (LeftArray, RightArray)
- Records "compare" operations during merging
- Records "overwrite" operations when copying elements back
- Demonstrates recursive structure with multiple record calls per merge

Teaching point: Students see how temporary arrays are used and how merging works element-by-element.

algorithms/quick.cpp

Implements Quick Sort with Lomuto partitioning. Key features:

- Records "pivot" operation to highlight the pivot element
- Records comparisons against the pivot
- Records swaps during partitioning
- Demonstrates in-place partitioning

Teaching point: The pivot remains highlighted throughout partitioning, making the partition process visually clear.

algorithms/heap.cpp

Implements Heap Sort with explicit heapify operations. Key features:

- Builds max-heap in-place
- Records comparisons during heapify (parent vs. children)
- Records swaps that maintain heap property
- Extracts maximum repeatedly

Teaching point: Students see the heap structure being maintained through swaps.

algorithms/shell.cpp

Implements Shell Sort with gap-based insertion. Key features:

- Uses gap sequence (n/2, n/4, ..., 1)
- Records comparisons across gaps
- Records shifts as operations
- Shows optimization over pure Insertion Sort

algorithms/radix.cpp

Implements Radix Sort (non-comparison). Key features:

- Uses counting sort for each digit
- Records "compare" during counting (even though it's not comparison-based, for visualization)
- Records "move_to_aux" when building output array
- Records "overwrite" when copying back
- Processes digits from least to most significant

Teaching point: Students see how a non-comparison sort works digit-by-digit.

algorithms/bucket.cpp

Implements Bucket Sort (distribution sort). Key features:

- Distributes elements into buckets based on value ranges
- Records "to_bucket" during distribution
- Uses `std::sort` within buckets (in practice, often Insertion Sort)
- Records "overwrite" when gathering results

Teaching point: Shows the power of distribution when data is uniformly distributed.

algorithms/SortEngine.cpp

The dispatcher. Contains only the `runAlgorithmByName()` function with a simple if-else chain. This centralized routing makes it easy to:

- Add new algorithms
- Handle invalid algorithm names
- Maintain consistent interfaces

step_recorder/StepRecorder.cpp

Manages the lifecycle of steps:

- `record()`: Appends a step to the vector
- `flushToFile()`: Serializes all steps to JSON
- `clear()`: Resets for a new sorting run

- `getSteps()`: Provides read access to the step vector

Uses the `nlohmann/json` library for all JSON operations.

include/Step.hpp

Defines the core data structures:

- `Step`: Represents one moment in algorithm execution
- `Metadata`: Summary information about the sorting run
- `Metrics`: Cumulative counters

These structs use C++17 features like `std::optional` for optional fields.

include/StepRecorder.hpp

Declares the `StepRecorder` class interface. Uses `nlohmann/json.hpp` for JSON types.

include/SortEngine.hpp

Declares all sorting algorithm functions as static methods. This header is included by each algorithm implementation file, providing access to the StepRecorder and related types.

main.cpp

The orchestrator. Responsibilities:

- Parse command-line arguments (algorithm, input file, output file)
- Load integer array from input file
- Create StepRecorder and Metrics objects
- Time the sorting operation
- Call SortEngine to execute the algorithm
- Build Metadata structure
- Flush results to JSON

Typical main.cpp flow:

```cpp
```

```cpp
int main(int argc, char* argv[]) {
    // 1. Parse arguments
    string alg = argv[1];
    string input = argv[2];
    string output = argv[3];

    // 2. Load data
    vector<int> arr = loadFromFile(input);

    // 3. Setup recording
    StepRecorder rec;
    Metrics metrics;

    // 4. Time and execute
    auto start = chrono::high_resolution_clock::now();
    SortEngine::runAlgorithmByName(alg, arr, rec, metrics);
    auto end = chrono::high_resolution_clock::now();

    // 5. Build metadata
    Metadata m;
    m.algorithm = alg;
    m.total_time_ms = duration_cast<milliseconds>(end - start).count();
    m.total_comparisons = metrics.comparisons;
    m.total_swaps = metrics.swaps;
    // ... set complexity strings based on algorithm

    // 6. Write output
    rec.flushToFile(output, m);

    return 0;
}
```

---

5. CMake & Build System

What is CMake?

CMake is a **build system generator**. Instead of writing platform-specific Makefiles (for Unix/Linux) or Visual Studio project files (for Windows), you write one `CMakeLists.txt` that works everywhere. CMake then generates the appropriate build files for your platform.

Why CMake for This Project?

This project uses external libraries (nlohmann/json). CMake handles:

- Downloading dependencies automatically
- Configuring include paths
- Compiling multiple source files
- Linking them into an executable
- Cross-platform compatibility (Windows, Linux, macOS)

CMakeLists.txt Structure (Conceptual)

cmake

```
cmake_minimum_required(VERSION 3.15)
project(SortingVisualizer)

# Fetch nlohmann/json library
include(FetchContent)
FetchContent_Declare(json
  URL https://github.com/nlohmann/json/releases/download/v3.11.2/json.tar.xz
)
FetchContent_MakeAvailable(json)

# Define executable
add_executable(sortgen
  main.cpp
  algorithms/bubble.cpp
  algorithms/selection.cpp
  # ... all other algorithm files
  algorithms/SortEngine.cpp
  step_recorder/StepRecorder.cpp
)

# Link JSON library
target_link_libraries(sortgen PRIVATE nlohmann_json::nlohmann_json)

# Set include directories
target_include_directories(sortgen PRIVATE include)
```

Build Process

Step 1: Configure

```
bash

cmake -S . -B build
```

This reads `CMakeLists.txt`, downloads dependencies, and generates build files in the `build/` directory.

Step 2: Compile

```
bash

cmake --build build
```

This compiles all `.cpp` files and links them into `sortgen.exe` (Windows) or `sortgen` (Unix).

Step 3: Run

```
bash

./build/sortgen bubble input.txt output.json
```

What CMake Produces

From the Makefile you provided, we can see CMake generated:

- Object files for each `.cpp` file (e.g., `bubble.obj`, `merge.obj`)
- A target named `sortgen` that links all objects
- Dependency tracking (if you modify one file, only that file and dependent files recompile)

The generated Makefile handles parallel compilation and incremental builds automatically.

Build Flags (Brief)

CMake can set compiler flags for optimization and debugging:

- `-O0 -g`: Debug mode (no optimization, include debug symbols)
- `-O3`: Release mode (maximum optimization)
- `-std=c++17`: Use C++17 standard features

---

6. Execution Flow

Let's walk through a complete execution from start to finish.

User Invocation

```bash
sortgen.exe merge dataset100.txt output_merge.json
```

Step-by-Step Internal Flow

1. Program Starts (main.cpp)

- `argc` = 4, `argv` = ["sortgen.exe", "merge", "dataset100.txt", "output_merge.json"]
- Parse arguments into variables

2. Load Dataset

```cpp
ifstream infile("dataset100.txt");
vector<int> arr;
int value;
while (infile >> value) {
    arr.push_back(value);
}
```

Now `arr` contains 100 integers.

3. Create Recording Infrastructure

```cpp
StepRecorder rec;
Metrics metrics; // comparisons = 0, swaps = 0
```

4. Start Timer

```cpp
auto start = chrono::high_resolution_clock::now();
```

5. Execute Algorithm

```cpp
SortEngine::runAlgorithmByName("merge", arr, rec, metrics);
```

Inside `runAlgorithmByName()`:

- The if-else chain matches "merge"
- Calls `mergeSort(arr, rec, metrics)`

Inside `mergeSort()`:

- Records initial state
- Calls `mergeSortRecursive()` with left=0, right=99
- Each recursive call:
  - Splits array in half
  - Recursively sorts left half
  - Recursively sorts right half
  - Merges the sorted halves (recording every comparison and overwrite)
- Returns to main

6. Stop Timer

```cpp
auto end = chrono::high_resolution_clock::now();
long long elapsed = duration_cast<milliseconds>(end - start).count();
```

7. Build Metadata

```cpp
```

```
Metadata m;
m.algorithm = "merge";
m.input_size = 100;
m.seed = /* extracted from input file or argv */;
m.start_time = getCurrentTimeISO8601();
m.total_time_ms = elapsed;
m.total_comparisons = metrics.comparisons;
m.total_swaps = metrics.swaps;
m.time_complexity = "O(n log n)";
m.space_complexity = "O(n)";
```

8. Flush to JSON

```
cpp

rec.flushToFile("output_merge.json", m);
```

The `StepRecorder` iterates through its `steps` vector, converts each Step to a JSON object, and writes the complete structure to the file.

9. Program Exits

```
cpp

return 0;
```

10. Frontend Visualization (Not Shown) The frontend application:

- Fetches `output_merge.json`
- Parses the JSON
- Renders the initial state
- Provides play/pause/step controls
- Animates through frames, coloring elements based on `highlights`

---

7. Key Design Patterns

Factory Pattern (Sort Engine)

The `SortEngine` uses a simple factory pattern—it manufactures algorithm executions based on string identifiers.

This is more flexible than hardcoding algorithm calls in main.

Observer Pattern (Step Recording)

The algorithms don't directly know about visualization—they just report events to the `StepRecorder`. The recorder accumulates these events. This separation of concerns means:

- Algorithms focus on sorting logic
- StepRecorder focuses on data capture
- main.cpp focuses on orchestration

Command Pattern (Steps)

Each `Step` is essentially a command object that describes an operation. The StepRecorder collects commands, and the frontend interprets them.

---

8. Why This Architecture?

Educational Clarity

Every component has a single, clear purpose:

- Algorithms sort and report
- StepRecorder captures
- main.cpp orchestrates
- JSON communicates

This makes the codebase easy to understand and modify.

Extensibility

Adding a new algorithm requires:

1. Create `algorithms/newalgorithm.cpp`
2. Implement the algorithm with recording calls
3. Add declaration to `SortEngine.hpp`
4. Add branch to `SortEngine.cpp`'s if-else chain
5. Recompile

No changes needed to StepRecorder, JSON serialization, or main.

Testability

Each algorithm can be tested independently:

```cpp
vector<int> test = {3, 1, 2};
StepRecorder rec;
Metrics m;
SortEngine::bubbleSort(test, rec, m);
// Check: test == {1, 2, 3}
// Check: rec.getSteps().size() > 0
```

Performance Considerations

This is NOT a performance-oriented design. It prioritizes:

- **Completeness**: Every operation is logged
- **Simplicity**: Easy to understand code
- **Pedagogical Value**: Students can see every detail

For production sorting, you'd never record every step—you'd just sort the array. But for learning, this overhead is worthwhile.

---

9. Common Student Questions

Q: Why copy the entire array in every step instead of just tracking changes?

A: Simplicity. The frontend can simply display `step.array` without reconstructing state. It's inefficient but pedagogically clear.

Q: Why not use inheritance for algorithms?

A: These algorithms are fundamentally different (recursive vs. iterative, comparison vs. distribution). A common interface would be forced and artificial. Static methods with uniform signatures provide the benefits of polymorphism without the complexity of inheritance.

Q: How big can the input be?

A: Practically, around 100-500 elements. Beyond that, JSON files become huge (10+ MB) and browser visualization slows down. This is a teaching tool, not a production system.

Q: Can I run the backend without a frontend?

A: Yes! The JSON output is human-readable. You can examine it in any text editor or use a JSON viewer online.

Q: What happens if I misspell the algorithm name?

A: `SortEngine::runAlgorithmByName()` throws a `runtime_error` with message "Unknown algorithm: [name]". main.cpp should catch this and print an error.

---

10. Debugging Tips

View Intermediate Steps

Add temporary print statements in algorithms:

```cpp
cout << "Comparing indices " << j << " and " << j+1 << endl;
```

Verify Step Count

Print `rec.getSteps().size()` before flushing. For Bubble Sort on n elements, expect roughly $O(n^2)$ steps.

Check JSON Validity

Use an online JSON validator (jsonlint.com) to ensure `flushToFile()` produces valid JSON.

Visualize Small Arrays

Test with 5-10 element arrays first. Manually trace the algorithm and compare your expectations with the JSON output.

Conclusion

This backend is an **educational instrumentation layer** wrapped around classic sorting algorithms. Every design choice—from comprehensive step recording to human-readable JSON—prioritizes learning over performance.

By understanding this architecture, you can:

- Explain how any algorithm is instrumented
- Modify algorithms to add new visualization features

- Add entirely new sorting algorithms
- Debug issues by inspecting JSON output
- Present the project confidently in lab demonstrations

The separation between sorting logic (algorithms/*.cpp), data capture (StepRecorder), orchestration (main.cpp), and communication (JSON) makes the system modular, maintainable, and most importantly, understandable for students learning both algorithms and systems programming.