Technical Documentation

1. File Structure

```
project-root/
├── cpp-core/                          (Backend Logic Engine)
│   ├── algorithms/
│   │   ├── bubble.cpp                 O(n²) sorting
│   │   ├── selection.cpp              O(n²) sorting
│   │   ├── insertion.cpp              O(n²) sorting
│   │   ├── merge.cpp                  O(n log n) divide-and-conquer
│   │   ├── quick.cpp                  O(n log n) partition-based
│   │   ├── heap.cpp                   O(n log n) heap-based
│   │   ├── shell.cpp                  Gap-based insertion sort
│   │   ├── radix.cpp                  Non-comparison digit sort
│   │   ├── bucket.cpp                 Distribution sort
│   │   └── SortEngine.cpp             Factory dispatcher
│   ├── step_recorder/
│   │   └── StepRecorder.cpp           Snapshot capture & JSON serialization
│   ├── include/
│   │   ├── Step.hpp                   Data structure definitions
│   │   ├── StepRecorder.hpp           Recorder interface
│   │   └── SortEngine.hpp             Algorithm declarations
│   ├── main.cpp                       CLI entry point
│   └── CMakeLists.txt                 Build configuration
│
├── frontend/                          (React Visualization UI)
│   ├── src/
│   │   ├── components/
│   │   │   ├── VisualizerCanvas.tsx   Canvas rendering
│   │   │   └── ControlPanel.tsx       Playback controls
│   │   ├── utils/
│   │   │   └── StepParser.ts          JSON validation
│   │   ├── types.ts                    TypeScript interfaces
│   │   └── App.tsx                     Main controller
│   ├── public/                         JSON logs storage
│   └── package.json                    Node dependencies
```

2. Core Data Structures

Step.hpp

Defines the snapshot structure capturing algorithm state at each operation.

```
struct Step {
    vector<int> array;                  // Complete array state
    vector<int> compare_indices;        // Elements being compared
    vector<int> swap_indices;           // Elements being swapped
    string operation;                   // "compare"|"swap"|"overwrite"|"pivot"|"none"
    size_t cumulative_comparisons;      // Total comparisons so far
    size_t cumulative_swaps;            // Total swaps so far
    optional<int> pivot_index;          // Quick Sort pivot tracker
    vector<int> sorted_indices;         // Elements in final position
};

struct Metadata {
    string algorithm;                   // Algorithm name
    size_t input_size;                  // Array size
    unsigned int seed;                  // Random seed for reproducibility
    string start_time;                  // ISO 8601 timestamp
    long long total_time_ms;            // Execution time
    size_t total_comparisons;           // Final comparison count
    size_t total_swaps;                 // Final swap count
    string time_complexity;             // Big-O notation
    string space_complexity;            // Space complexity
};

struct Metrics {
    size_t comparisons = 0;             // Running comparison counter
    size_t swaps = 0;                   // Running swap counter
};
```

---

3. Backend Implementation

3.1 main.cpp

Orchestrates the entire backend pipeline: argument parsing, data loading, algorithm execution, timing, and JSON output.

```
int main(int argc, char* argv[]) {
    // Parse command-line arguments
    string algorithm = parseArg(argv, "--algorithm");
    int size = parseArg(argv, "--size");
    string output = parseArg(argv, "--output");

    // Generate or load input array
    vector<int> arr = generateRandomArray(size);

    // Initialize recording infrastructure
    StepRecorder recorder;
    Metrics metrics;

    // Execute algorithm with timing
    auto start = chrono::high_resolution_clock::now();
    SortEngine::runAlgorithmByName(algorithm, arr, recorder, metrics);
    auto end = chrono::high_resolution_clock::now();

    // Build metadata
    Metadata meta;
    meta.algorithm = algorithm;
    meta.input_size = size;
    meta.total_time_ms = duration_cast<milliseconds>(end - start).count();
    meta.total_comparisons = metrics.comparisons;
    meta.total_swaps = metrics.swaps;

    // Serialize to JSON
    recorder.flushToFile(output, meta);

    return 0;
}
```

3.2 SortEngine.cpp

Factory pattern dispatcher routing algorithm name strings to implementation functions.

```
void SortEngine::runAlgorithmByName(const string& name, vector<int>& arr,
                                    StepRecorder& rec, Metrics& metrics) {
    if (name == "bubble") {
        bubbleSort(arr, rec, metrics);
    } else if (name == "selection") {
        selectionSort(arr, rec, metrics);
    } else if (name == "insertion") {
        insertionSort(arr, rec, metrics);
    } else if (name == "merge") {
        mergeSort(arr, rec, metrics);
    } else if (name == "quick") {
        quickSort(arr, rec, metrics);
    } else if (name == "heap") {
        heapSort(arr, rec, metrics);
    } else if (name == "shell") {
        shellSort(arr, rec, metrics);
    } else if (name == "radix") {
        radixSort(arr, rec, metrics);
    } else if (name == "bucket") {
        bucketSort(arr, rec, metrics);
    } else {
        throw runtime_error("Unknown algorithm: " + name);
    }
}
```

3.3 StepRecorder.cpp

Accumulates snapshots and serializes to JSON using nlohmann/json library.

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

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

    // Serialize metadata
    j["metadata"] = {
        {"algorithm", m.algorithm},
        {"input_size", m.input_size},
        {"seed", m.seed},
        {"start_time", m.start_time},
        {"total_time_ms", m.total_time_ms},
        {"total_comparisons", m.total_comparisons},
        {"total_swaps", m.total_swaps},
        {"time_complexity", m.time_complexity},
        {"space_complexity", m.space_complexity}
    };

    // Serialize frames
    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);
    }

    // Write to file with indentation
    ofstream file(path);
    file << j.dump(2);
}
```

4. Algorithm Implementations

4.1 Bubble Sort (bubble.cpp)

Adjacent element comparison with early termination optimization.

```
void SortEngine::bubbleSort(vector<int>& arr, StepRecorder& rec, Metrics& metrics) {
    size_t n = arr.size();
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});

    for (size_t i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (size_t j = 0; j < n - i - 1; j++) {
            // Record comparison before comparing
            metrics.comparisons++;
            rec.record({arr, {j, j+1}, {}, "compare",
                        metrics.comparisons, metrics.swaps});

            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                metrics.swaps++;
                swapped = true;
                // Record swap after swapping
                rec.record({arr, {j, j+1}, {j, j+1}, "swap",
                            metrics.comparisons, metrics.swaps});
            }
        }
        if (!swapped) break;  // Early termination
    }

    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
}
```

4.2 Selection Sort (selection.cpp)

Linear scan for minimum element, single swap per pass.

```
void SortEngine::selectionSort(vector<int>& arr, StepRecorder& rec, Metrics& metrics) {
    size_t n = arr.size();
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});

    for (size_t i = 0; i < n - 1; i++) {
        size_t min_idx = i;

        // Find minimum in unsorted portion
        for (size_t j = i + 1; j < n; j++) {
            metrics.comparisons++;
            rec.record({arr, {min_idx, j}, {}, "compare",
                       metrics.comparisons, metrics.swaps});

            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap if needed
        if (min_idx != i) {
            swap(arr[min_idx], arr[i]);
            metrics.swaps++;
            rec.record({arr, {min_idx, i}, {min_idx, i}, "swap",
                       metrics.comparisons, metrics.swaps});
        }
    }

    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
}
```

4.3 Insertion Sort (insertion.cpp)

Builds sorted array by inserting elements into correct positions with shifts.

```
void SortEngine::insertionSort(vector<int>& arr, StepRecorder& rec, Metrics& metrics) {
    size_t n = arr.size();
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});

    for (size_t i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Shift elements greater than key
        while (j >= 0) {
            metrics.comparisons++;
            rec.record({arr, {j, j+1}, {}, "compare",
                        metrics.comparisons, metrics.swaps});

            if (arr[j] > key) {
                arr[j + 1] = arr[j];
                metrics.swaps++;
                rec.record({arr, {j, j+1}, {j+1}, "overwrite",
                            metrics.comparisons, metrics.swaps});
                j--;
            } else {
                break;
            }
        }

        // Insert key in correct position
        arr[j + 1] = key;
        rec.record({arr, {j+1}, {j+1}, "overwrite",
                    metrics.comparisons, metrics.swaps});
    }

    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
}
```

4.4 Merge Sort (merge.cpp)

Divide-and-conquer with auxiliary arrays for merging.

```
void merge(vector<int>& arr, int left, int mid, int right,
           StepRecorder& rec, Metrics& metrics) {
    // Create temporary subarrays
    vector<int> L(arr.begin() + left, arr.begin() + mid + 1);
    vector<int> R(arr.begin() + mid + 1, arr.begin() + right + 1);

    int i = 0, j = 0, k = left;

    // Merge subarrays
    while (i < L.size() && j < R.size()) {
        metrics.comparisons++;
        rec.record({arr, {left+i, mid+1+j}, {}, "compare",
                    metrics.comparisons, metrics.swaps});

        if (L[i] <= R[j]) {
            arr[k] = L[i++];
        } else {
            arr[k] = R[j++];
        }
        rec.record({arr, {k}, {k}, "overwrite",
                    metrics.comparisons, metrics.swaps});
        k++;
    }

    // Copy remaining elements
    while (i < L.size()) {
        arr[k++] = L[i++];
        rec.record({arr, {k-1}, {k-1}, "overwrite",
                    metrics.comparisons, metrics.swaps});
    }
    while (j < R.size()) {
        arr[k++] = R[j++];
        rec.record({arr, {k-1}, {k-1}, "overwrite",
                    metrics.comparisons, metrics.swaps});
    }
}

void mergeSortRecursive(vector<int>& arr, int left, int right,
                        StepRecorder& rec, Metrics& metrics) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSortRecursive(arr, left, mid, rec, metrics);
    mergeSortRecursive(arr, mid + 1, right, rec, metrics);
    merge(arr, left, mid, right, rec, metrics);
}

void SortEngine::mergeSort(vector<int>& arr, StepRecorder& rec, Metrics& metrics) {
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
    mergeSortRecursive(arr, 0, arr.size() - 1, rec, metrics);
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
}
```

4.5 Quick Sort (quick.cpp)

Lomuto partition scheme with explicit pivot tracking.

```
int partition(vector<int>& arr, int low, int high,
              StepRecorder& rec, Metrics& metrics) {
    int pivot = arr[high];
    int i = low - 1;

    // Highlight pivot
    rec.record({arr, {high}, {}, "pivot",
                metrics.comparisons, metrics.swaps, high});

    // Partition around pivot
    for (int j = low; j <= high - 1; j++) {
        metrics.comparisons++;
        rec.record({arr, {j, high}, {}, "compare",
                    metrics.comparisons, metrics.swaps, high});

        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
            metrics.swaps++;
            rec.record({arr, {i, j}, {i, j}, "swap",
                        metrics.comparisons, metrics.swaps, high});
        }
    }

    // Place pivot in correct position
    swap(arr[i + 1], arr[high]);
    metrics.swaps++;
    rec.record({arr, {i+1, high}, {i+1, high}, "swap",
                metrics.comparisons, metrics.swaps, high});

    return i + 1;
}

void quickSortRecursive(vector<int>& arr, int low, int high,
                        StepRecorder& rec, Metrics& metrics) {
    if (low < high) {
        int pi = partition(arr, low, high, rec, metrics);
        quickSortRecursive(arr, low, pi - 1, rec, metrics);
        quickSortRecursive(arr, pi + 1, high, rec, metrics);
    }
}

void SortEngine::quickSort(vector<int>& arr, StepRecorder& rec, Metrics& metrics) {
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
    quickSortRecursive(arr, 0, arr.size() - 1, rec, metrics);
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
}
```

4.6 Heap Sort (heap.cpp)

Max-heap construction followed by repeated extraction.

```
void heapify(vector<int>& arr, int n, int i,
             StepRecorder& rec, Metrics& metrics) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // Find largest among root, left, right
    if (left < n) {
        metrics.comparisons++;
        rec.record({arr, {left, largest}, {}, "compare",
                    metrics.comparisons, metrics.swaps});
        if (arr[left] > arr[largest]) largest = left;
    }

    if (right < n) {
        metrics.comparisons++;
        rec.record({arr, {right, largest}, {}, "compare",
                    metrics.comparisons, metrics.swaps});
        if (arr[right] > arr[largest]) largest = right;
    }

    // Swap and recursively heapify if needed
    if (largest != i) {
        swap(arr[i], arr[largest]);
        metrics.swaps++;
        rec.record({arr, {i, largest}, {i, largest}, "swap",
                    metrics.comparisons, metrics.swaps});
        heapify(arr, n, largest, rec, metrics);
    }
}

void SortEngine::heapSort(vector<int>& arr, StepRecorder& rec, Metrics& metrics) {
    int n = arr.size();
    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});

    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i, rec, metrics);

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        metrics.swaps++;
        rec.record({arr, {0, i}, {0, i}, "swap",
                    metrics.comparisons, metrics.swaps});
        heapify(arr, i, 0, rec, metrics);
    }

    rec.record({arr, {}, {}, "none", metrics.comparisons, metrics.swaps});
}
```

5. JSON Output Schema

Complete structure of generated log files consumed by the frontend.


```
{
  "metadata": {
    "algorithm": "quick",
    "input_size": 50,
    "seed": 12345,
    "start_time": "2025-12-21T10:00:00Z",
    "total_time_ms": 42,
    "total_comparisons": 120,
    "total_swaps": 45,
    "time_complexity": "O(n log n)",
    "space_complexity": "O(log n)"
  },
  "frames": [
    {
      "array": [10, 5, 2, 8, 15, 3, 7, 12, ...],
      "operation": "compare",
      "cumulative_comparisons": 5,
      "cumulative_swaps": 2,
      "highlights": {
        "compare": [0, 1],
        "swap": [],
        "pivot": 4
      }
    },
    {
      "array": [5, 10, 2, 8, 15, 3, 7, 12, ...],
      "operation": "swap",
      "cumulative_comparisons": 5,
      "cumulative_swaps": 3,
      "highlights": {
        "compare": [],
        "swap": [0, 1],
        "pivot": 4
      }
    }
  ]
}
```

6. Build System

6.1 CMakeLists.txt

Cross-platform build configuration using CMake with automatic dependency management.

```
cmake_minimum_required(VERSION 3.15)
project(SortingVisualizer VERSION 1.0)

# Set C++17 standard
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# 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)

# Collect all algorithm source files
file(GLOB ALGORITHM_SOURCES "algorithms/*.cpp")

# Define executable
add_executable(sortgen
    main.cpp
    ${ALGORITHM_SOURCES}
    step_recorder/StepRecorder.cpp
)

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

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

6.2 Compilation Steps

```
# Configure build
cd cpp-core
mkdir build
cd build
cmake ..

# Compile
cmake --build .

# Verify
./sortgen --help
```

---

7. CLI Usage

7.1 Command Syntax

```
./sortgen --algorithm <name> --size <n> --output <file.json> [--seed <value>]
```

7.2 Available Algorithms

`bubble`, `selection`, `insertion`, `merge`, `quick`, `heap`, `shell`, `radix`, `bucket`

7.3 Examples

```
# Generate Bubble Sort log
./sortgen --algorithm bubble --size 50 --output bubble_50.json

# Generate Quick Sort with seed
./sortgen --algorithm quick --size 100 --output quick.json --seed 999

# Generate Merge Sort
./sortgen --algorithm merge --size 30 --output merge_30.json
```