

# Sorting Visualizer - Presentation Notes

## SORTING ALGORITHMS LEARNING GUIDE

### 1. Bubble Sort

Bubble sort is like bubbles rising in a glass of soda: larger elements "bubble up" to the end through repeated swaps of adjacent items. In plain C++, bubble sort uses two nested loops. The outer loop runs  $n-1$  times, and the inner loop compares adjacent elements from the start to the end minus the outer loop's progress. If two neighbors are out of order, swap them. Add a flag to stop early if no swaps happen.

Time: Best  $O(n)$  (sorted), average/worst  $O(n^2)$ . Space:  $O(1)$ . Stable. Comparison-based.

In this project (bubble.cpp), we insert `rec.record` calls before each comparison and after each swap. These just snapshot the state and metrics but don't alter the logic—the array sorts the same way.

### 2. Selection Sort

Imagine selecting the smallest gift from a pile repeatedly: scan the unsorted part for the minimum, then swap it to the front. In C++, an outer loop tracks the sorted prefix. For each position  $i$ , find the min index from  $i+1$  to end by scanning and comparing. Swap if needed.

Time: Always  $O(n^2)$  (fixed scans). Space:  $O(1)$ . Unstable. Comparison-based.

Project mods (selection.cpp): Record "compare" before each min check. Record "swap" after actual swap. Metrics update comparisons/swaps. Logging observes without changing—min finding and swaps proceed identically.

### 3. Insertion Sort

Like sorting playing cards: insert each new card into its place in the sorted hand by shifting others. In C++, build a sorted prefix. For each element  $i$  from 1 to  $n-1$ , store it as key, shift larger elements right while comparing backward.

Time: Best  $O(n)$ , average/worst  $O(n^2)$ . Space:  $O(1)$ . Stable. Comparison-based.

In insertion.cpp, record "compare" in the while loop before shifting. Use "overwrite" for shifts instead of swap. Final key placement also "overwrite". These records capture shifts visually but logic remains pure insertion.

### 4. Merge Sort

Divide a problem into halves, solve them, then merge—like sorting two piles of papers by repeatedly taking the smaller top card. In C++, recursively split array until single elements, then merge sorted halves by comparing and copying to temp, then back.

Time: Always  $O(n \log n)$ . Space:  $O(n)$  (temp arrays). Stable. Comparison-based.

merge.cpp adds records in merge(): "compare" before picking from left/right, "overwrite" when placing into array. Recursion unchanged. Logging shows subarray merges without affecting divide-conquer.

## 5. Quick Sort

Pick a "pivot" like a random card, partition deck into less/greater, recurse—like quick team sorting by height around a median. In C++ (Lomuto scheme), choose last as pivot, partition by swapping smaller to left. Recurse on subarrays.

Time: Best/average  $O(n \log n)$ , worst  $O(n^2)$  (bad pivots). Space:  $O(\log n)$  avg (recursion). Unstable. Comparison-based.

quick.cpp records "pivot" highlight, "compare" in partition loop, "swap" when moving. Recursion as is. Instrumentation tracks partitioning visually, correctness intact.

## 6. Heap Sort

Build a heap (priority queue) like a tournament bracket, extract max repeatedly. In C++, build max-heap by heapify down from bottom. Then swap root to end, heapify reduced heap.

Time:  $O(n \log n)$ . Space:  $O(1)$ . Unstable. Comparison-based.

heap.cpp records "compare" in heapify ifs, "swap" if needed. Build and extract phases logged. Adds visibility to heap operations without change.

## 7. Shell Sort

Gap-insertion sort: start with large gaps like jumping chess pieces, reduce to refine. In C++, gaps halve from  $n/2$ . For each gap, insertion-sort sublists.

Time: Depends on gaps, avg  $O(n \log n)$ , worst  $O(n^2)$ . Space:  $O(1)$ . Unstable. Comparison-based.

shell.cpp mirrors insertion but with gaps. Records "compare" and "swap" in inner loop. Gap reduction logged implicitly via steps.

## 8. Radix Sort

Sort by digits like mail by zip code: group by least significant digit, then next. In C++ (for ints), use counting sort per digit (1,10,100...).

Time:  $O(d \times n)$  ( $d$  digits). Space:  $O(n + k)$  ( $k=10$ ). Stable. Non-comparison (distribution).

radix.cpp uses counting sort loops, records "move\_to\_aux" when bucketing, "overwrite" when copying back. Digit passes visualized.

## 9. Bucket Sort

Scatter into buckets like sorting exam scores into grade bins, sort bins, gather. In C++ (for ints), compute bucket

index by value range, sort each (e.g., `std::sort`), concatenate.

Time: Avg  $O(n)$ , worst  $O(n^2)$  if one bucket. Space:  $O(n)$ . Stable if bin sorts stable. Non-comparison.

`bucket.cpp` records "overwrite" when gathering back. Distribution and bin sorts visualized via steps.

---

## BACKEND ENGINE & LOGGING GUIDE

### Overall Backend Architecture

The backend is a command-line tool (`sortgen.exe`) that runs sorts and logs steps. `main.cpp` is the boss: it parses user args, generates or loads an array, creates a `StepRecorder`, picks the algorithm via `SortEngine`, times it, and flushes JSON. `SortEngine.cpp` is the dispatcher: `runAlgorithmByName` uses if-else to call the right static method based on name string. This lets runtime selection without recompiling. Algorithms live in separate files under `algorithms/`, keeping code clean.

### Step Recording System

A "step" or "snapshot" is like a photo of the array at a key moment: what it looks like, which indices are active, the operation type, and running totals of comparisons/swaps. We record EVERY comparison and swap because visualization needs fine-grained playback—students see exactly where things happen. In code, algorithms call `rec.record(Step{...})` with current array copy, indices, operation, cumulatives. `Metrics` struct tracks counts, incremented before recording. JSON files are large because each step duplicates the array. That's intentional: frontend needs full states for independent frame rendering, no recomputation.

### JSON Logging

JSON is chosen because it's easy to parse in JavaScript (frontend) and human-readable for debugging. Structure: "metadata" contains algorithm name, size, seed, time taken, totals, complexities. "frames" is array of objects, each with "array" (current values), "highlights" (compare/swap/pivot indices), "operation", cumulatives. Each frame lets frontend draw one animation step. `StepRecorder.cpp` accumulates steps in a vector, then in `flushToFile` builds `nlohmann::json`, serializes with indentation.

### File Responsibilities

`algorithms/*.cpp`: Each has a static void function taking array ref, recorder ref, metrics ref. Implements algo with interspersed `rec.record` calls. For recursive sorts, helpers handle subcalls. `StepRecorder.cpp`: Pushes steps, clears, flushes to file. `include/ headers`: `Step.hpp` defines `Step/Metadata/Metrics` structs. `StepRecorder.hpp` class interface. `SortEngine.hpp` declares functions. `main.cpp`: Handles CLI, sets up array/seed, times execution, fills metadata, calls flush.

### CMake & Build System

CMake is a build tool that generates makefiles for your compiler. Here, `CMakeLists.txt` sets C++17, fetches

JSON lib, includes dirs, globs algo sources, builds sortgen executable, links JSON. Why CMake? Cross-platform builds, handles dependencies automatically. Run cmake in build dir, then build to get sortgen.exe.

## **Execution Flow**

User runs sortgen with algorithm, size, output args. main.cpp parses args. Generate array: random shuffle with seed or load from file. Create recorder and metrics. Start timer, call SortEngine::runAlgorithmByName which dispatches to the algorithm. Algorithm runs, recording steps. End timer, fill metadata. Recorder flushes JSON to path.