Sorting Algorithms Learning Guide

Introduction

This guide explains each sorting algorithm implemented in the Interactive Sorting Algorithm Visualizer. We'll build your understanding from the ground up—starting with intuition, walking through examples, analyzing complexity, and finally connecting to the actual implementation.

---

1. Bubble Sort

The Intuition

Imagine you have a line of students arranged by height, but they're in random order. Bubble Sort works like repeatedly walking down the line and swapping adjacent students if they're out of order. After each pass, the tallest remaining unsorted student "bubbles up" to their correct position at the end.

How It Works (Standard C++)

Bubble Sort compares adjacent elements and swaps them if they're in the wrong order. It makes multiple passes through the array until no more swaps are needed.

```cpp
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped) break; // Already sorted
    }
}
```

Step-by-Step Example

Let's sort: **[5, 2, 8, 1, 9]**

Pass 1:

- Compare 5 and 2 → Swap → [2, 5, 8, 1, 9]
- Compare 5 and 8 → No swap
- Compare 8 and 1 → Swap → [2, 5, 1, 8, 9]
- Compare 8 and 9 → No swap
- Result: 9 is now in correct position

Pass 2:

- Compare 2 and 5 → No swap
- Compare 5 and 1 → Swap → [2, 1, 5, 8, 9]
- Compare 5 and 8 → No swap
- Result: 8 is now in correct position

Pass 3:

- Compare 2 and 1 → Swap → [1, 2, 5, 8, 9]
- Compare 2 and 5 → No swap
- Result: Array is sorted!

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n) — when array is already sorted with optimization
  - Average case: $O(n^2)$ — typical random data
  - Worst case: $O(n^2)$ — when array is reverse sorted
- **Space Complexity:** O(1) — only uses a few variables
- **Stability:** Stable — equal elements maintain their relative order
- **Type:** Comparison-based

Project Implementation Modifications

In `bubble.cpp`, the algorithm is modified to support visualization:

1. **Before the comparison**, we record a snapshot showing which two elements are being compared:

```cpp
rec.record({arr, {j, j+1}, {}, "compare", ...});
```

2. **After a swap**, we record another snapshot showing the swap happened:

```cpp
rec.record({arr, {j, j+1}, {j, j+1}, "swap", ...});
```

3. **Metrics tracking**: The algorithm increments `metrics.comparisons` for every comparison and `metrics.swaps` for every swap.

These recording calls do NOT change the algorithm's correctness—they simply create a history of states for the frontend to animate. The sorting logic remains identical.

---

2. Selection Sort

The Intuition

Think of organizing a deck of cards in your hand. Selection Sort works by repeatedly finding the smallest card from the unsorted portion and placing it at the beginning. In the first pass, you find the absolute smallest and put it first. In the second pass, you find the second smallest and put it second, and so on.

How It Works (Standard C++)

Selection Sort divides the array into sorted and unsorted regions. It repeatedly selects the minimum element from the unsorted region and swaps it with the first unsorted element.

```cpp
```

```
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        if (min_idx != i) {
            swap(arr[min_idx], arr[i]);
        }
    }
}
```

Step-by-Step Example

Let's sort: **[64, 25, 12, 22, 11]**

Pass 1: Find minimum in [64, 25, 12, 22, 11]

- Minimum is 11 at index 4
- Swap with position 0 → **[11, 25, 12, 22, 64]**

Pass 2: Find minimum in [25, 12, 22, 64]

- Minimum is 12 at index 2
- Swap with position 1 → **[11, 12, 25, 22, 64]**

Pass 3: Find minimum in [25, 22, 64]

- Minimum is 22 at index 3
- Swap with position 2 → **[11, 12, 22, 25, 64]**

Pass 4: Find minimum in [25, 64]

- Minimum is 25 (already in place)
- No swap needed → **[11, 12, 22, 25, 64]**

Complexity Analysis

- **Time Complexity:**
  - Best case: $O(n^2)$ — always scans entire unsorted portion
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$
- **Space Complexity:** O(1) — in-place sorting
- **Stability:** Unstable — can change relative order of equal elements
- **Type:** Comparison-based

Key Insight: Selection Sort makes fewer swaps than Bubble Sort (at most n-1), but still makes $O(n^2)$ comparisons.

Project Implementation Modifications

In `selection.cpp`, the visualization tracking includes:

1. **During the search for minimum**: Each comparison is recorded:

```cpp
rec.record({arr, {min_idx, j}, {}, "compare", ...});
```

2. **When the minimum is found and swapped**: The swap is recorded:

```cpp
rec.record({arr, {min_idx, i}, {min_idx, i}, "swap", ...});
```

The frontend can highlight which element is currently considered the minimum, making the selection process visually clear.

---

3. Insertion Sort

The Intuition

Imagine sorting a hand of playing cards. You pick up cards one at a time and insert each into its correct position among the cards you're already holding. Insertion Sort works the same way—it builds the sorted array one element at a time by inserting each new element into its proper place.

How It Works (Standard C++)

Insertion Sort maintains a sorted portion at the beginning of the array. For each unsorted element, it finds the correct position in the sorted portion and inserts it there.

```cpp
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // Shift right
            j--;
        }
        arr[j + 1] = key; // Insert key
    }
}
```

Step-by-Step Example

Let's sort: **[12, 11, 13, 5, 6]**

Step 1: key = 11

- Compare 11 with 12 → 11 < 12, shift 12 right
- Insert 11 → **[11, 12, 13, 5, 6]**

Step 2: key = 13

- Compare 13 with 12 → 13 > 12, already in place
- **[11, 12, 13, 5, 6]**

Step 3: key = 5

- Compare 5 with 13 → shift 13 right
- Compare 5 with 12 → shift 12 right
- Compare 5 with 11 → shift 11 right
- Insert 5 → **[5, 11, 12, 13, 6]**

Step 4: key = 6

- Compare 6 with 13 → shift 13 right
- Compare 6 with 12 → shift 12 right
- Compare 6 with 11 → shift 11 right
- Compare 6 with 5 → 6 > 5, stop
- Insert 6 → **[5, 6, 11, 12, 13]**

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n) — when array is already sorted
  - Average case: $O(n^2)$
  - Worst case: $O(n^2)$ — when array is reverse sorted
- **Space Complexity:** O(1) — in-place sorting
- **Stability:** Stable — maintains relative order of equal elements
- **Type:** Comparison-based

Key Advantage: Very efficient for small datasets and nearly sorted arrays.

Project Implementation Modifications

In `insertion.cpp`, the implementation tracks shifts as overwrites:

1. **Each comparison** while searching for the insertion position:

```
cpp

  rec.record({arr, {j, j+1}, {}, "compare", ...});
```

2. **Each shift operation** (moving element right):

```
cpp

  rec.record({arr, {j, j+1}, {j+1}, "overwrite", ...});
```

3. **Final insertion** of the key:

```cpp
rec.record({arr, {j+1}, {j+1}, "overwrite", ...});
```

Note that shifts are visualized as "overwrite" operations rather than swaps, which accurately represents what's happening in memory.

---

4. Merge Sort

The Intuition

Imagine you have two already-sorted piles of papers and need to merge them into one sorted pile. You'd repeatedly take the smaller top paper from either pile and add it to your result pile. Merge Sort works by recursively dividing the array in half until you have single-element arrays (which are trivially sorted), then merging those sorted pieces back together.

How It Works (Standard C++)

Merge Sort is a divide-and-conquer algorithm with two main phases:

1. **Divide:** Recursively split the array in half
2. **Conquer:** Merge sorted subarrays back together

```cpp
```

```
void merge(vector<int>& arr, int left, int mid, int right) {
    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;
    while (i < L.size() && j < R.size()) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    while (i < L.size()) arr[k++] = L[i++];
    while (j < R.size()) arr[k++] = R[j++];
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Step-by-Step Example

Let's sort: **[38, 27, 43, 3]**

Divide Phase:

```
[38, 27, 43, 3]
   ↓
[38, 27]  [43, 3]
   ↓       ↓
[38] [27]  [43] [3]
```

Merge Phase:

```
[38] [27] → Compare 38 and 27 → [27, 38]
[43] [3]  → Compare 43 and 3  → [3, 43]

[27, 38] [3, 43]
   ↓
Compare 27 and 3  → Take 3  → [3, ...]
Compare 27 and 43 → Take 27 → [3, 27, ...]
Compare 38 and 43 → Take 38 → [3, 27, 38, ...]
Only 43 left      → Take 43 → [3, 27, 38, 43]
```

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n log n)
  - Average case: O(n log n)
  - Worst case: O(n log n)
  - **Guaranteed O(n log n)** — one of its main advantages
- **Space Complexity:** O(n) — needs temporary arrays for merging
- **Stability:** Stable — equal elements maintain their relative order
- **Type:** Comparison-based

Project Implementation Modifications

In `merge.cpp`, the recursive structure is preserved with added visualization:

1. **During comparisons** between left and right subarrays:

```cpp
rec.record({arr, {left+i, mid+1+j}, {}, "compare", ...});
```

2. **When copying elements** into the merged position:

```cpp
rec.record({arr, {k}, {k}, "overwrite", ...});
```

The visualization shows which elements from the auxiliary arrays are being compared and how they're merged back into the original array. This helps students understand the merge process clearly.

---

5. Quick Sort

The Intuition

Imagine organizing books on a shelf by picking one book as a reference (the "pivot"), then arranging all books: those alphabetically before it on the left, those after it on the right. Then recursively do the same for the left and right sections. Quick Sort uses this "partition-and-conquer" strategy.

How It Works (Standard C++)

Quick Sort picks a pivot element and partitions the array so that all elements smaller than the pivot come before it, and all larger elements come after it. Then it recursively sorts the two partitions.

cpp

```cpp
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Choose last element as pivot
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]); // Place pivot
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Step-by-Step Example

Let's sort: **[10, 80, 30, 90, 40, 50, 70]** using last element as pivot

First Partition (pivot = 70):

- i = -1, scan array
- 10 < 70 → i=0, swap arr[0] with arr[0] → [10, 80, 30, 90, 40, 50, 70]
- 80 > 70 → no swap
- 30 < 70 → i=1, swap arr[1] with arr[2] → [10, 30, 80, 90, 40, 50, 70]
- 90 > 70 → no swap
- 40 < 70 → i=2, swap arr[2] with arr[4] → [10, 30, 40, 90, 80, 50, 70]
- 50 < 70 → i=3, swap arr[3] with arr[5] → [10, 30, 40, 50, 80, 90, 70]
- Place pivot: swap arr[4] with arr[6] → [10, 30, 40, 50, 70, 90, 80]

Now 70 is in correct position! Recursively sort [10, 30, 40, 50] and [90, 80].

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n log n) — balanced partitions
  - Average case: O(n log n)
  - Worst case: $O(n^2)$ — already sorted with bad pivot choice
- **Space Complexity:** O(log n) — recursion stack
- **Stability:** Unstable — relative order can change
- **Type:** Comparison-based

Key Advantage: Very fast in practice with good cache performance.

Project Implementation Modifications

In `quick.cpp`, the Lomuto partition scheme is used with visualization:

1. **Pivot selection** is highlighted:

```cpp
rec.record({arr, {high}, {}, "pivot", ...});
```

2. **Each comparison** with the pivot:

```cpp
rec.record({arr, {j, high}, {}, "compare", ...});
```

3. **Each swap** during partitioning:

```cpp
rec.record({arr, {i, j}, {i, j}, "swap", ...});
```

The pivot index is passed through the Step structure so the frontend can continuously highlight which element is the current pivot during partitioning.

---

6. Heap Sort

The Intuition

Think of a tournament bracket where the winner (maximum element) always rises to the top. Heap Sort builds a max-heap data structure where the largest element is always at the root. It repeatedly extracts the maximum and rebuilds the heap, building the sorted array from right to left.

How It Works (Standard C++)

Heap Sort uses a binary max-heap represented as an array. A max-heap satisfies the property that each parent is greater than or equal to its children.

```cpp
```

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

    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();

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

    // Extract elements from heap
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]); // Move max to end
        heapify(arr, i, 0);    // Restore heap
    }
}
```

Step-by-Step Example

Let's sort: **[4, 10, 3, 5, 1]**

Build Max Heap:

```
Initial: [4, 10, 3, 5, 1]

Heapify from index 1 (value 10):
- Check children: 5 and 1
- 10 > 5 and 10 > 1, no change

Heapify from index 0 (value 4):
- Check children: 10 and 3
- 10 > 4, swap → [10, 4, 3, 5, 1]
- Recursively heapify index 1
- Check children of 4: 5 and 1
- 5 > 4, swap → [10, 5, 3, 4, 1]

Max Heap: [10, 5, 3, 4, 1]
```

Extract and Sort:

- Swap 10 with 1 → [1, 5, 3, 4, | 10]
- Heapify → [5, 4, 3, 1, | 10]
- Swap 5 with 1 → [1, 4, 3, | 5, 10]
- Heapify → [4, 1, 3, | 5, 10]
- Continue until sorted: [1, 3, 4, 5, 10]

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n log n)
  - Average case: O(n log n)
  - Worst case: O(n log n)
  - **Guaranteed O(n log n)**
- **Space Complexity:** O(1) — in-place sorting
- **Stability:** Unstable
- **Type:** Comparison-based

Project Implementation Modifications

In `heap.cpp`, the heapify process is visualized:

1. **Comparisons** between parent and children:

```cpp
rec.record({arr, {left, largest}, {}, "compare", ...});
```

2. **Swaps** during heapification:

```cpp
rec.record({arr, {i, largest}, {i, largest}, "swap", ...});
```

Students can see how the heap property is maintained after each extraction, making this complex algorithm more understandable.

---

7. Shell Sort

The Intuition

Shell Sort is like an enhanced Insertion Sort. Instead of comparing only adjacent elements, it compares elements that are far apart (using a "gap"). As the gap decreases to 1, elements get closer to their final positions. This reduces the total number of shifts needed.

How It Works (Standard C++)

Shell Sort uses a gap sequence. It performs gapped insertion sorts, gradually reducing the gap until it becomes 1 (regular insertion sort).

```cpp
```

```
void shellSort(vector<int>& arr) {
    int n = arr.size();

    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}
```

Step-by-Step Example

Let's sort: **[35, 33, 42, 10, 14, 19, 27, 44]**

Gap = 4:

- Compare elements 4 positions apart
- 35 vs 14 → swap → [14, 33, 42, 10, 35, 19, 27, 44]
- 33 vs 19 → swap → [14, 19, 42, 10, 35, 33, 27, 44]
- etc.
- Result: [14, 19, 27, 10, 35, 33, 42, 44]

Gap = 2:

- Compare elements 2 positions apart
- Further organize → [10, 14, 27, 19, 35, 33, 42, 44]

Gap = 1:

- Regular insertion sort on nearly sorted array
- Final: [10, 14, 19, 27, 33, 35, 42, 44]

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n log n)
  - Average case: Depends on gap sequence, typically O(n^(3/2))
  - Worst case: $O(n^2)$
- **Space Complexity:** O(1)
- **Stability:** Unstable
- **Type:** Comparison-based

Project Implementation Modifications

In `shell.cpp`, gap-based comparisons are tracked:

```cpp
rec.record({arr, {i, i-gap}, {}, "compare", ...});
```

The visualization shows elements being compared across gaps, helping students understand why Shell Sort is faster than pure Insertion Sort on larger arrays.

---

8. Radix Sort

The Intuition

Imagine sorting a stack of papers by date. Instead of comparing entire dates, you could first group by year, then within each year by month, then by day. Radix Sort does this with numbers—it sorts digit by digit, from least significant to most significant.

How It Works (Standard C++)

Radix Sort processes each digit position using a stable sort (typically counting sort). It starts from the least significant digit and moves to the most significant.

```cpp
```

```
void radixSort(vector<int>& arr) {
    int maxVal = *max_element(arr.begin(), arr.end());

    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countingSortByDigit(arr, exp);
    }
}
```

Step-by-Step Example

Let's sort: **[170, 45, 75, 90, 802, 24, 2, 66]**

Sort by 1s digit (exp = 1):

- Extract last digits: [0, 5, 5, 0, 2, 4, 2, 6]
- Stable sort: [170, 90, 802, 2, 24, 45, 75, 66]

Sort by 10s digit (exp = 10):

- Extract 10s digits: [7, 9, 0, 0, 2, 4, 7, 6]
- Stable sort: [802, 2, 24, 45, 66, 170, 75, 90]

Sort by 100s digit (exp = 100):

- Extract 100s digits: [8, 0, 0, 0, 0, 1, 0, 0]
- Stable sort: [2, 24, 45, 66, 75, 90, 170, 802]

Complexity Analysis

- **Time Complexity:** O(d × (n + k))
    - d = number of digits
    - k = range of each digit (10 for decimal)
    - Linear for fixed d
- **Space Complexity:** O(n + k) — temporary arrays
- **Stability:** Stable
- **Type:** Non-comparison-based (distribution sort)

Key Advantage: Can be faster than O(n log n) comparison sorts when d is small.

Project Implementation Modifications

In `radix.cpp`, each digit processing pass is visualized:

```cpp
rec.record({arr, {i}, {}, "compare", ...}); // Counting phase
rec.record({arr, {i}, {}, "move_to_aux", ...}); // Moving to output
rec.record({arr, {}, {i}, "overwrite", ...}); // Copying back
```

Students can see how the array becomes progressively more sorted after each digit pass.

---

9. Bucket Sort

The Intuition

Imagine sorting mail by ZIP code. You'd first put all mail into buckets based on ZIP code ranges, then sort within each bucket. Bucket Sort divides the input into buckets based on value ranges, sorts each bucket individually, and concatenates them.

How It Works (Standard C++)

Bucket Sort distributes elements into buckets, sorts each bucket (often using another algorithm), and merges the results.

```cpp

```

```
void bucketSort(vector<int>& arr) {
    int n = arr.size();
    int maxVal = *max_element(arr.begin(), arr.end());
    int minVal = *min_element(arr.begin(), arr.end());

    int bucketCount = n;
    vector<vector<int>> buckets(bucketCount);
    double range = (double)(maxVal - minVal + 1) / bucketCount;

    // Distribute into buckets
    for (int i = 0; i < n; i++) {
        int idx = (arr[i] - minVal) / range;
        if (idx >= bucketCount) idx = bucketCount - 1;
        buckets[idx].push_back(arr[i]);
    }

    // Sort buckets and gather
    int idx = 0;
    for (auto& bucket : buckets) {
        sort(bucket.begin(), bucket.end());
        for (int val : bucket) {
            arr[idx++] = val;
        }
    }
}
```

Step-by-Step Example

Let's sort: **[42, 32, 33, 52, 37, 47, 51]** with 4 buckets

Range calculation:

- Min = 32, Max = 52, Range per bucket = (52-32+1)/4 ≈ 5.25

Distribution:

- Bucket 0 [32-37): [32, 33]
- Bucket 1 [37-42): [37]
- Bucket 2 [42-47): [42, 47]
- Bucket 3 [47-52]: [52, 47, 51]

Sort each bucket:

- Bucket 0: [32, 33]
- Bucket 1: [37]
- Bucket 2: [42, 47]
- Bucket 3: [47, 51, 52]

Concatenate:

- Result: [32, 33, 37, 42, 47, 47, 51, 52]

Complexity Analysis

- **Time Complexity:**
  - Best case: O(n + k) — uniform distribution
  - Average case: O(n + k)
  - Worst case: O(n^2) — all elements in one bucket
- **Space Complexity:** O(n + k)
- **Stability:** Can be stable depending on bucket sorting method
- **Type:** Non-comparison-based (distribution sort)

Project Implementation Modifications

In `bucket.cpp`, the distribution and gathering phases are visualized:

```cpp
rec.record({arr, {i}, {}, "to_bucket", ...}); // Distribution
rec.record({arr, {}, {idx}, "overwrite", ...}); // Gathering
```

Students can see how elements are distributed into buckets and then gathered back in sorted order.

Summary Table

Algorithm	Time (Avg)	Time (Worst)	Space	Stable	Type
Bubble	O(n^2)	O(n^2)	O(1)	Yes	Comparison
Selection	O(n^2)	O(n^2)	O(1)	No	Comparison

Algorithm	Time (Avg)	Time (Worst)	Space	Stable	Type
Insertion	$O(n^2)$	$O(n^2)$	O(1)	Yes	Comparison
Merge	O(n log n)	O(n log n)	O(n)	Yes	Comparison
Quick	O(n log n)	$O(n^2)$	O(log n)	No	Comparison
Heap	O(n log n)	O(n log n)	O(1)	No	Comparison
Shell	O(n^1.5)	$O(n^2)$	O(1)	No	Comparison
Radix	O(d×n)	O(d×n)	O(n+k)	Yes	Distribution
Bucket	O(n+k)	$O(n^2)$	O(n+k)	Can be	Distribution

This guide has taught you each algorithm conceptually before introducing the implementation details. Remember, the logging code in this project doesn't change the algorithms' correctness—it simply captures their execution for educational visualization.