https://brain-mentors.com

https://skillrisers.com

Cycle Sort - Complete Guide for Beginners

Created for: College Freshers and Programming Beginners
Focus: Understanding Cycle Sort algorithm with minimal writes

Table of Contents

1. Introduction to Cycle Sort

What is Cycle Sort?

Cycle Sort is an in-place, unstable sorting algorithm that is particularly useful when the cost of writing to memory is very high. It minimizes the number of memory writes by placing each element directly in its correct position.

Real-world Analogy:

Imagine Suresh works at Indian Railways in Chennai and needs to arrange train coaches in the correct order (Coach 1, Coach 2, Coach 3, etc.). However, moving coaches on the track is very expensive and time-consuming. Instead of moving coaches multiple times like in Bubble Sort, Suresh finds the exact final position for each coach and moves it only once to that position. This is exactly what Cycle Sort does!

Key Characteristics

- **In-place sorting**: Requires O(1) extra space
- **Unstable**: Does not preserve relative order of equal elements
- **Minimal writes**: Each element is written to memory at most once (except when reading)
- **Optimal for expensive writes**: Best when write operations are costly
- **Not optimal for comparisons**: Makes O(n²) comparisons

---

2. Why Cycle Sort is Special

The Problem with Other Sorting Algorithms

Most sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort may swap elements multiple times before reaching their final position.

Example with Bubble Sort:

```
Array: [5, 1, 4, 2, 3]

Element 1 moves through:
Position 1 → Position 0 (2 writes)

Element 5 moves through:
Position 0 → Position 1 → Position 2 → Position 3 → Position 4 (5 writes)

Total writes for these 2 elements: 7 writes
```

The Cycle Sort Solution

Cycle Sort finds the exact final position of each element and places it there directly.

Example with Cycle Sort:

```
Array: [5, 1, 4, 2, 3]

Element 5: Calculate correct position = 4
Write once at position 4 (1 write)

Element 1: Calculate correct position = 0
Write once at position 0 (1 write)

Total writes for these 2 elements: 2 writes
```

This makes Cycle Sort optimal when:

- Writing to memory is expensive (Flash memory, EEPROM)
- You want to minimize disk writes
- Physical movement of items is costly (like train coaches!)

---

3. The Core Concept

Understanding "Cycles"

The name "Cycle Sort" comes from the fact that elements form cycles when determining their correct positions.

What is a Cycle?

A cycle is a sequence of positions where:

- Element at position A should go to position B
- Element at position B should go to position C
- Element at position C should go to position A (completing the cycle)

Simple Example:

```
Array: [3, 1, 2]

Position:  0  1  2
Element:   3  1  2

Cycle:
- Element 3 (at position 0) belongs at position 2
- Element 2 (at position 2) belongs at position 1
- Element 1 (at position 1) belongs at position 0
- Back to position 0 (cycle complete!)

Cycle: 0 → 2 → 1 → 0
```

Finding Correct Position

To find where an element should go:

1. Count how many elements are smaller than it
2. That count is the correct position (for ascending order)

Example:

```
Array: [5, 2, 8, 1, 9]
Element: 5

How many elements are smaller than 5?
- 2 is smaller ✓
- 8 is not smaller
- 1 is smaller ✓
- 9 is not smaller

Count = 2
So, 5 should be at position 2 (0-indexed)
```

4. How Cycle Sort Works

Algorithm Steps

1. **Start a cycle**: Begin with the first position (0)
2. **Find correct position**: Count elements smaller than current element
3. **Place element**: Put current element at its correct position
4. **Continue cycle**: Take the element displaced and find its position
5. **Complete cycle**: When we return to starting position, cycle is complete
6. **Next cycle**: Move to next position and repeat
7. **Finish**: When all positions are covered, array is sorted

Step-by-Step Process

Let's understand with array: `[5, 2, 4, 3, 1]`

Cycle 1: Starting at position 0 (element 5)

Step 1: Count elements smaller than 5

- 2 < 5 ✓
- 4 < 5 ✓
- 3 < 5 ✓
- 1 < 5 ✓
  Count = 4, so position = 4

Step 2: Place 5 at position 4, take 1 (was at position 4)

Step 3: Now process 1, count elements smaller than 1

- No elements smaller than 1
  Count = 0, so position = 0

Step 4: Place 1 at position 0 (cycle complete!)

Array after Cycle 1: `[1, 2, 4, 3, 5]`

Cycle 2: Starting at position 1 (element 2)

Element 2 is already in correct position (no cycle needed)

Cycle 3: Starting at position 2 (element 4)

Step 1: Count elements smaller than 4 (in remaining positions)

- 3 < 4 ✓
  Count = 1 (from position 2), so position = 3

Step 2: Place 4 at position 3, take 3

Step 3: Process 3, count elements smaller than 3 (from position 2)

- No elements smaller
  Position = 2

Step 4: Place 3 at position 2 (cycle complete!)

Array after Cycle 3: `[1, 2, 3, 4, 5]`

Result: Sorted!

---

5. Visual Representation

Visual Diagram - Complete Process

```
Initial Array: [5, 2, 4, 3, 1]
Index:          0  1  2  3  4

══════════════════════════════════════════════════════════
CYCLE 1: Starting at position 0
══════════════════════════════════════════════════════════

Step 1: Pick element at position 0
   ┌─┐
   │5│ 2  4  3  1
   └─┘
   Position: 0

Step 2: Count elements smaller than 5
   5  [2] [4] [3] [1]  ← All 4 elements are smaller
       ✓   ✓   ✓   ✓

   Correct position for 5 = 4

Step 3: Place 5 at position 4, pick displaced element
   _  2  4  3  ┌─┐
               │5│
   ┌─┐         └─┘
   │1│ ← Picked from position 4
   └─┘

Step 4: Count elements smaller than 1
   1  [2] [4] [3]  ← No elements smaller

   Correct position for 1 = 0

Step 5: Place 1 at position 0 (back to start - cycle complete!)
   ┌─┐
   │1│ 2  4  3  5
   └─┘
   ✓ Cycle 1 complete!

Array after Cycle 1: [1, 2, 4, 3, 5]
```

```
════════════════════════════════════════════════════════════
CYCLE 2: Starting at position 1
════════════════════════════════════════════════════════════

Step 1: Pick element at position 1
        ┌─┐
   1  │2│ 4  3  5
        └─┘

Step 2: Count elements smaller than 2
   [1] 2  [4] [3] [5]  ← Only 1 is smaller (but it's before position 1)
    ✓

   From position 1 onwards: no elements smaller
   Correct position for 2 = 1 (already correct!)

   ✓ No cycle needed, element already in place

Array after Cycle 2: [1, 2, 4, 3, 5]

════════════════════════════════════════════════════════════
CYCLE 3: Starting at position 2
════════════════════════════════════════════════════════════

Step 1: Pick element at position 2
            ┌─┐
   1  2   │4│ 3  5
            └─┘

Step 2: Count elements smaller than 4 (from position 2 onwards)
   [1] [2] 4  [3] [5]  ← 3 is smaller
                ✓

   Correct position for 4 = 3 (position 2 + 1)

Step 3: Place 4 at position 3, pick displaced element
   1  2  _  ┌─┐ 5
             │4│
             └─┘
   ┌─┐
   │3│ ← Picked from position 3
   └─┘

Step 4: Count elements smaller than 3 (from position 2 onwards)
   [1] [2] 3  [4] [5]  ← No elements smaller

   Correct position for 3 = 2

Step 5: Place 3 at position 2 (back to start - cycle complete!)
            ┌─┐
   1  2   │3│ 4  5
            └─┘

   ✓ Cycle 3 complete!
```

```
Array after Cycle 3: [1, 2, 3, 4, 5]

═══════════════════════════════════════════════════════════
CYCLE 4 & 5: Starting at positions 3 and 4
═══════════════════════════════════════════════════════════

Both elements (4 and 5) are already in correct positions
No cycles needed

═══════════════════════════════════════════════════════════
FINAL SORTED ARRAY: [1, 2, 3, 4, 5]
═══════════════════════════════════════════════════════════
```

Cycle Detection Visualization

```
Example: [4, 3, 2, 1]

Cycle 1: Position 0 (element 4)
    ┌──────────────┐
    ↓              │
  ┌───┐          ┌───┐
  │ 4 │ → ... → │ 1 │
  └───┘          └───┘
  Pos 0          Pos 3
    ↑              ↓
    │            ┌───┐
    │            │ 2 │
    │            └───┘
    │            Pos 2
    │              ↓
    │            ┌───┐
    └──────────→ │ 3 │
                 └───┘
                 Pos 1

Cycle path: 0 → 3 → 2 → 1 → 0
Elements: 4 → 1 → 2 → 3 → 4 (back to start)
```

6. Complete Dry Run

Example 1: [5, 2, 4, 3, 1]

```
Initial Array: [5, 2, 4, 3, 1]
Positions:      0  1  2  3  4
```

Cycle 1: Starting at index 0

Step	Current Element	Current Pos	Count Smaller	Correct Pos	Action	Array State
1	5	0	4 (2,4,3,1)	4	Place 5 at pos 4	[_, 2, 4, 3, 5]
2	1	4	0	0	Place 1 at pos 0	[1, 2, 4, 3, 5]
3	-	0	-	-	Back to start, cycle complete	[1, 2, 4, 3, 5]

Writes in Cycle 1: 2

Cycle 2: Starting at index 1

Step	Current Element	Current Pos	Count Smaller	Correct Pos	Action	Array State
1	2	1	0 (from pos 1)	1	Already correct	[1, 2, 4, 3, 5]

Writes in Cycle 2: 0

Cycle 3: Starting at index 2

Step	Current Element	Current Pos	Count Smaller	Correct Pos	Action	Array State
1	4	2	1 (3 is smaller)	3	Place 4 at pos 3	[1, 2, _, 4, 5]
2	3	3	0 (from pos 2)	2	Place 3 at pos 2	[1, 2, 3, 4, 5]
3	-	2	-	-	Back to start, cycle complete	[1, 2, 3, 4, 5]

Writes in Cycle 3: 2

Cycles 4 & 5: Elements already in place

Total Writes: 4 (compared to 10-15 in other algorithms!)

Example 2: [3, 1, 5, 2, 4] - Detailed Trace

```
Initial Array: [3, 1, 5, 2, 4]
Positions:      0  1  2  3  4
```

Complete Step-by-Step Table

Cycle	Start Pos	Element	Count	Target Pos	Displaced	Action	Array After
1	**0**	**3**	**2** (1,2 smaller)	**2**	**5**	Write 3→pos 2	[_, 1, 3, 2, 4]
1 cont	2	5	2 (from pos 0: 1,2,3,4)	4	4	Write 5→pos 4	[_, 1, 3, 2, 5]
1 cont	4	4	3 (from pos 0: 1,2,3)	3	2	Write 4→pos 3	[_, 1, 3, 4, 5]
1 cont	3	2	1 (from pos 0: 1)	1	1	Write 2→pos 1	[_, 2, 3, 4, 5]
1 cont	1	1	0	0	-	Write 1→pos 0	[1, 2, 3, 4, 5]
1 end	-	-	-	-	-	Cycle complete	[1, 2, 3, 4, 5]
2	**1**	**2**	-	**1**	-	Already correct	[1, 2, 3, 4, 5]
3	**2**	**3**	-	**2**	-	Already correct	[1, 2, 3, 4, 5]
4	**3**	**4**	-	**3**	-	Already correct	[1, 2, 3, 4, 5]
5	**4**	**5**	-	**4**	-	Already correct	[1, 2, 3, 4, 5]

Statistics:

- **Total Cycles Started**: 5
- **Actual Cycles Performed**: 1 (others were already in place)
- **Total Writes**: 5
- **Total Comparisons**: 10

Example 3: Array with Duplicates [4, 3, 2, 3, 1]

```
Initial Array: [4, 3, 2, 3, 1]
Positions:      0  1  2  3  4
```

Handling Duplicates

Cycle	Position	Element	Count	Target	Note	Array After
1	0	4	4	4	All smaller	[_, 3, 2, 3, 4]
1	4	1	0	0	None smaller	[1, 3, 2, 3, 4]
2	1	3	2	3	2,1 smaller, skip duplicate 3	[1, _, 2, 3, 4]
2	3	3	2	3	**Duplicate!** Skip to next	[1, _, 2, 3, 4]
2	3	3	2	3+1=4	Adjust position	[1, _, 2, _, 4]
2	4	4	-	-	Already processed	[1, _, 2, 3, 4]
2	1	3	2	2	Place at adjusted position	[1, 3, 2, 3, 4]

Important: When encountering duplicates at target position, increment position to find next available spot.

---

7. Java Implementation

Basic Implementation

```java
import java.util.Arrays;

public class CycleSort {

    public static void cycleSort(int[] arr) {
        int n = arr.length;

        for (int cycleStart = 0; cycleStart < n - 1; cycleStart++) {

            int item = arr[cycleStart];
            int pos = cycleStart;

            // Find correct position
            for (int i = cycleStart + 1; i < n; i++) {
                if (arr[i] < item) {
                    pos++;
                }
            }

            // If already in correct position
            if (pos == cycleStart) {
                continue;
            }

            // Skip duplicates
            while (item == arr[pos]) {
                pos++;
            }

            // Put item to correct position
            if (pos != cycleStart) {
                int temp = arr[pos];
                arr[pos] = item;
                item = temp;
            }

            // Rotate the rest of the cycle
            while (pos != cycleStart) {
                pos = cycleStart;

                for (int i = cycleStart + 1; i < n; i++) {
                    if (arr[i] < item) {
                        pos++;
                    }
                }

                while (item == arr[pos]) {
                    pos++;
                }

                int temp = arr[pos];
                arr[pos] = item;
                item = temp;
            }
        }
    }

    public static void main(String[] args) {
        int[] policies = {101, 105, 102, 104, 103};

        System.out.println("Before:");
        System.out.println(Arrays.toString(policies));

        cycleSort(policies);

        System.out.println("After:");
        System.out.println(Arrays.toString(policies));
    }
}
```

---

8. Time and Space Complexity

Time Complexity Analysis

Best Case: O(n²)

- Even if array is already sorted
- Still need to check position for each element
- Cannot be optimized like Bubble Sort

Average Case: O(n²)

- For each of n elements
- Compare with remaining n elements
- $n \times n = n^2$ comparisons

Worst Case: $O(n^2)$

- Reverse sorted array
- Maximum number of cycles
- Still $O(n^2)$ comparisons

Space Complexity

O(1) - Constant extra space

- Only uses a few variables (item, pos, temp)
- All sorting done in-place
- No additional arrays needed

Comparison Count vs Write Count

Comparisons: $O(n^2)$

- For each element: compare with all elements to its right
- Not optimal for comparisons

Writes: O(n)

- Each element written at most once to its final position
- **Optimal for writes!** This is the key advantage

Comparison with Other Algorithms

Algorithm	Comparisons	Writes	Space
Cycle Sort	$O(n^2)$	**O(n)**	O(1)
Selection Sort	$O(n^2)$	O(n)	O(1)
Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(log n)

Key Insight: Cycle Sort has the **minimum number of writes** of any sorting algorithm!

---

9. Advantages and Disadvantages

Advantages

1. **Minimum Writes**

  - Each element written at most once
  - Optimal when write operations are expensive
  - Perfect for Flash memory, EEPROM, or mechanical storage

2. **In-place Sorting**

   - Requires O(1) extra space
   - No additional arrays needed
   - Memory efficient

3. **Predictable Performance**

   - Always $O(n^2)$ time complexity
   - No best/worst case variations in write count
   - Reliable for critical systems

4. **Useful for Specific Hardware**

   - Flash memory (limited write cycles)
   - EEPROM (expensive writes)
   - Systems where writes cost more than reads

Real-world Example:
Ramesh works at a Pune manufacturing unit where products are stored on an expensive automated conveyor system. Moving products is costly (each movement costs electricity and wear). Cycle Sort ensures each product is moved only once to its correct position, minimizing operational costs.

Disadvantages

1. **High Comparison Count**

   - $O(n^2)$ comparisons in all cases
   - Not efficient if comparisons are expensive
   - Slower than O(n log n) algorithms for large data

2. **Unstable**

   - Does not preserve relative order of equal elements
   - Not suitable when stability is required

3. **Complex Implementation**

   - More difficult to understand than simple sorts
   - Easy to make mistakes with cycle logic
   - Requires careful handling of duplicates

4. **No Early Exit**

   - Cannot optimize for sorted arrays
   - Always performs same number of operations
   - Unlike Bubble Sort which can detect sorted state

5. **Not Cache Friendly**
    - Jumps around in memory
    - Poor locality of reference
    - Can be slower in practice despite fewer writes