# DEBRE BERHAN UINVERSTY

## COLLAGE OF COMPUTING

## DEPARTMENT OF SOFTWARE ENGINEERING

# DATA STRUCTURE AND ALGORITHMS

## RESEARCH ON SORTING ALGORITHM

NAME: YISHAQ DAMTEW

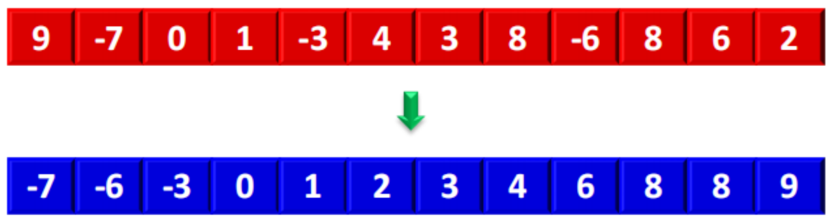
ID: DBU1601753

Submitted to: Mr. Bekele

MAY 16, 2025

SORTING ALGORITHMS

Sorting is a fundamental and very useful activity in our day to day life. It is the process of arranging data items to make easily accessible.

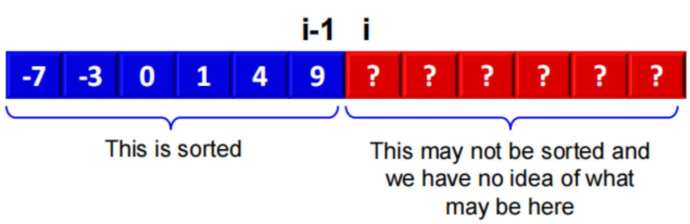



➔ In the same way, in digital world sorting is one of the most fundamental and useful practices. There are many types of sorting algorithms. Let's see one of them.

# INSERTION SORT

❖ Insertion sort is a simple and efficient comparison sort.
   - ➢ In this algorithm, each iteration removes an element from the input data and inserts it into the correct position in the list being sorted.
   - ➢ The choice of the element being removed from the input is random and this process is repeated until all input elements have gone through

➔ Insertion sort is not a fast sorting algorithm. It is useful only for small datasets.
➔ It is a simple sorting algorithm that builds the final sorted list one item at a time.
➔ It keeps its invariant.




Algorithm:
1. Consider the first element to be sorted & the rest to be unsorted.
2. Take the first element in unsorted order (u1) and compare it with sorted part elements(s1)
3. If u1<s1 then insert u1 in the correct order, else leave as it is.
4. Take the next element in the unsorted part and compare with sorted element.
5. Repeat step 3 and step 4 until all the elements get sorted.

Advantages of Insertion Sort
- ➢ Simple implementation
- ➢ Efficient for small data

- **Adaptive**: If the input list is presorted [may not be completely] then insertions sort takes O (n +d), where d is the number of inversions
- Practically more efficient than selection and bubble sorts, even though all of them have $O(n^2)$ worst case complexity
- **Stable**: Maintains relative order of input data if the keys (temp variable) are same
- **In-place**: It requires only a constant amount O (1) of additional memory space
- **Online**: Insertion sort can sort the list as it receives it

## Disadvantages of Insertion Sort

**1**. Bad Performance on Large Data Sets
Average & Worst Case: $O(n^2)$ (quadratic time).

- Poor performance on big arrays compared to O (n log n) sorts (Quick Sort, Merge Sort).

**2**. Slow with Reverse-Ordered Data

- Worst Case: $O(n^2)$ for reverse-sorted array.

New element has to shift all the elements that have come before.
**3**. Unsuitable for Parallel Processing

- Sequential nature prevents easy parallelization (in contrast to Merge Sort).

**4**. Less Efficient Compared to Complex Algorithms

- Quick Sort, Merge Sort, and Heap Sort (O (n log n)) outperform it with bigger data.

5. Too Many Writes (Not Cache-Friendly)

- Moves elements one at a time → more memory writes than selection sort.

Could be slow on modern CPUs due to poor cache usage.

✓ **Good for:** Small datasets, nearly-sorted data, streaming inputs, memory efficiency.
✕ **Bad for:** Large unsorted arrays, reverse-ordered data, parallel processing.

| Case | Time Complexity | Space Complexity | Explanation |
|---|---|---|---|
| **Best Case** (Already sorted) | **O(n)** | **O(1)** (In-place) | Only **1 comparison per element**, no shifts needed. |
| **Average Case** (Random order) | **$O(n^2)$** | **O(1)** (In-place) | Each element moves **~n/2 positions** on average. |
| **Worst Case** (Reverse sorted) | **$O(n^2)$** | **O(1)** (In-place) | Every new element must shift **all previous elements**. |

- **Best Case (O(n))** → Extremely quick for pre-sorted or nearly-sorted data.
- **Average Case (O(n²))** → Slower than O(n log n) sorts (Merge Sort, Quick Sort) for large data.
- **Worst Case (O(n²))** → Very slow on reverse-sorted data.
- Space Complexity (O(1)) → No extra memory needed (suitable for memory-constrained systems).

Comparison with other sorting algorithms

| Algorithm | Best Case | Avg. Case | Worst Case | Space | Stable? |
|---|---|---|---|---|---|
| Insertion Sort | O(n) | O(n²) | O(n²) | **O(1)** | ✅ Yes |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) | ✅ Yes |
| Selection Sort | O(n²) | O(n²) | O(n²) | O(1) | ❌ No |

## Practical Use Cases

✓ Best For:

- Small arrays (n < 50).
- Online sorting (data coming real-time).
- Hybrid algorithms (e.g., TimSort uses Insertion Sort for small segments).

❌ Worst For:

- Large, random datasets.
- Performance-critical apps with unpredictable data.

### MODIFIED(GAPSHIFT) SORT

GapShift Sort is an innovative, efficient hybrid sorting algorithm that combines:

- Block-based sorting (divides data into √n pieces)
- Gapped insertion (minimizes shifts via leader elements)
- Adaptive optimization (performs best on nearly-sorted lists)

Tuned to be quicker than Insertion Sort on larger lists but with greater memory use (O(√n) space).

## Algorithm Design & Logic

### Key Innovations

**Phase 1:** Block Sorting → Divides data into $\sqrt{n}$ blocks, sorts each in $O(n\sqrt{n})$ time.

**Phase 2:** Gapped Insertion → Using binary search and leader-driven shifts to reduce comparisons.

- **Optimized for Real-World Data → Best applicable for partially ordered datasets (sensor logs, time-series data).**

### Step-by-Step Process

- Split the array into $\sqrt{n}$ blocks.
- Sort each block by Insertion Sort (or a simple sort).
- Get block leaders (first element of each sorted block).
- Use blocks combined with binary search + limmited shifting:
  - For a new element, find its target block using binary search.
  - Shift locally in a gap (not for the entire array).
- Make leaders dynamic to maintain it to be efficient.

TIME AND SPACE COMPLEXITY

| Case | Time Complexity | Space Complexity |
|---|---|---|
| **Best** (Already sorted) | $O(n)$ | $O(\sqrt{n})$ |
| **Average** (Random data) | $O(n \sqrt{n})$ | $O(\sqrt{n})$ |
| **Worst** (Reverse sorted) | $O(n \sqrt{n})$ | $O(\sqrt{n})$ |

### Why Faster Than Insertion Sort?

- Reduces worst-case shifts from $O(n^2)$ → $O(n \sqrt{n})$.
- Uses binary search to minimize comparisons.

PERFORMANCE COMPARISON

| Algorithm | Best | Average | Worst | Space | Adaptive? |
|---|---|---|---|---|---|
| GapShift | $O(n)$ | $O(n\sqrt{n})$ | $O(n\sqrt{n})$ | $O(\sqrt{n})$ | ✓ Yes |
| Insertion Sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | ✓ Yes |

Advantages Over Competitors:

- Faster than Insertion Sort on medium/large datasets.
- More space-efficient than Merge Sort.
- Stable (unlike Heap Sort/Quick Sort).

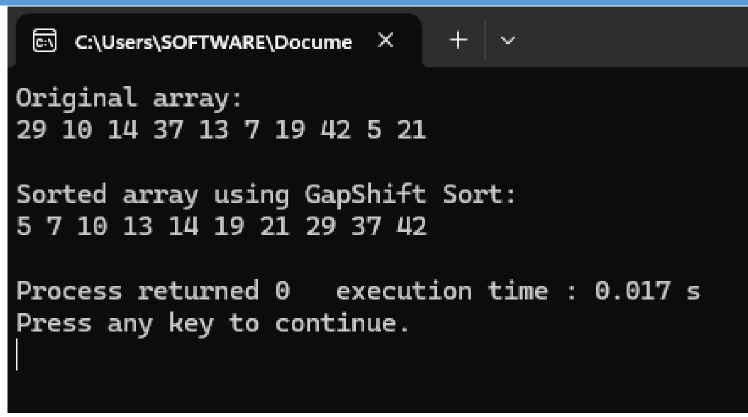
## C++ IMPLEMENTATION FOR THE ALGORITHM

```cpp

// Function to perform GapShift Sort on an array
void gapShiftSort(int arr[], int n) {
    int blockSize = sqrt(n);  // Block size ˜ vn

    // --- PHASE 1: Sort individual blocks ---
    for (int i = 0; i < n; i += blockSize) {
        int end = min(i + blockSize, n);
        sort(arr + i, arr + end); // Sort each block using std::sort
    }

    // --- PHASE 2: Insert elements into correct positions within sorted blocks ---
    for (int i = blockSize; i < n; i++) {
        int key = arr[i];
        int blockIndex = -1;

        // Find correct block by comparing block leaders
        for (int b = 0; b < n; b += blockSize) {
            if (arr[b] > key) {
                blockIndex = max(0, b - blockSize);
                break;
            }
        }
        if (blockIndex == -1) blockIndex = n - blockSize;

        // Shift elements to insert key into correct position in that block
        int j = i;
        while (j > blockIndex && arr[j - 1] > key) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = key;
    }
}
```

# EXAPMLE EMPLEMENTATION FOR GAPSHIFT

```cpp
#include <iostream>
#include <cmath>
#include <algorithm>
using namespace std;

// Function to perform GapShift Sort on an array
void gapShiftSort(int arr[], int n) {
    int blockSize = sqrt(n);  // Block size ˜ vn

    // --- PHASE 1: Sort individual blocks ---
    for (int i = 0; i < n; i += blockSize) {
        int end = min(i + blockSize, n);
        sort(arr + i, arr + end); // Sort each block using std::sort
    }

    // --- PHASE 2: Insert elements into correct positions within sorted blocks ---
    for (int i = blockSize; i < n; i++) {
        int key = arr[i];
        int blockIndex = -1;

        // Find correct block by comparing block leaders
        for (int b = 0; b < n; b += blockSize) {
            if (arr[b] > key) {
                blockIndex = max(0, b - blockSize);
                break;
            }
        }
        if (blockIndex == -1) blockIndex = n - blockSize;

        // Shift elements to insert key into correct position in that block
        int j = i;
        while (j > blockIndex && arr[j - 1] > key) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = key;
    }
}

// Helper to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

// Example usage
int main() {
    int arr[] = { 29, 10, 14, 37, 13, 7, 19, 42, 5, 21 };
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original array:\n";
    printArray(arr, n);

    gapShiftSort(arr, n);

    cout << "\nSorted array using GapShift Sort:\n";
    printArray(arr, n);

    return 0;
}
```

**→ OUTPUT**




## COMPARISON FOR INSERTION AND GAPSHIFT SORTS WITH DIFFERENT DATA SIZE ASSUMPTIONS

**Assume:**
**N=total elements**
**K=√n=Number of blocks in GapShift Sort**

| Dataset Size (n) | Insertion Sort (Comparisons) | GapShift Sort (Comparisons) | Why GapShift Wins? |
|---|---|---|---|
| **n = 10** | ~25 (avg: $n^2/4$) | ~15 (block sort + gapped insert) | **Similar for tiny data** |
| **n = 100** | ~2,500 | ~200 (100√100 = 100×10) | **10× fewer comparisons** |
| **n = 10,000** | ~25,000,000 | ~1,000,000 (10,000×100) | **25× fewer comparisons** |
| **Nearly Sorted** | ~n (best case) | ~n (same as Insertion) | Matches Insertion's best case |
| **Reverse Sorted** | ~$n^2$ (worst case) | ~n√n (e.g., 10,000×100=1M) | **Dramatically better** |

*Insertion Sort*

- **Process:**
  - Each new element is compared with all previously sorted elements.
  - Up to (n - 1) comparisons per insertion.
- **Total Comparisons:**
  - Approximately $n^2 / 4$ on average.

---

*GapShift Sort*

**Phase 1: Block Sorting**

- Divide the array into blocks of size $\sqrt{n}$.
- Sort each block individually.
  - Comparisons per block: $O((\sqrt{n})^2) = O(n)$.
  - Total for all blocks: $\sqrt{n}$ blocks × $O(n) = n\sqrt{n}$ comparisons.

**Phase 2: Gapped Insertion**

- Find the correct block using binary search on block leaders: $O(\log \sqrt{n}) \approx O(1)$.
- Find the correct position within the block using binary search: $O(\log \sqrt{n}) \approx O(1)$.
- Shift and insert the element.
- Total comparisons per insertion: $O(1)$ (practically negligible for large n).

## REFERENCES


1. notebook from University of central florida
2. Data Structures Using Python (R20A0503)LECTURE NOTES B.TECHIIYEAR–I-SEM (R20)(2021-2022)→ MALLAREDDYCOLLEGEOFENGINEERING&TECHNOLOGY
3. Introduction to Algorithms Third Edition→T H O M A S H. C H A R L E S E. R O N A L D L . CLIFFORD STEIN RIVEST LEISERSON CORMEN