5. Simple Sorting Algorithms

5.1 Bubble Sort

Concept

Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they're in the wrong order.

How it works:

1. Compare adjacent elements
2. Swap if in wrong order
3. Repeat for all elements
4. After each pass, largest element "bubbles" to the end

Visual Example:

```
Initial: [64, 34, 25, 12, 22, 11, 90]

Pass 1:
[34, 64, 25, 12, 22, 11, 90]  → 64 > 34, swap
[34, 25, 64, 12, 22, 11, 90]  → 64 > 25, swap
[34, 25, 12, 64, 22, 11, 90]  → 64 > 12, swap
[34, 25, 12, 22, 64, 11, 90]  → 64 > 22, swap
[34, 25, 12, 22, 11, 64, 90]  → 64 > 11, swap
[34, 25, 12, 22, 11, 64, 90]  → 64 < 90, no swap
→ 90 is in correct position!

Pass 2:
[25, 34, 12, 22, 11, 64, 90]
...continues until sorted...

Final: [11, 12, 22, 25, 34, 64, 90]
```

Implementation

Basic Bubble Sort:

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Last i elements are already in place
        for (int j = 0; j < n - i - 1; j++) {
            // Swap if element is greater than next
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Optimized Bubble Sort (with flag):

```
void bubbleSortOptimized(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int swapped = 0;  // Flag to detect if any swap happened

        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = 1;
            }
        }

        // If no swaps, array is sorted
        if (swapped == 0) {
            break;
        }
    }
}
```

Why Optimization Helps:

```
Nearly sorted array: [11, 12, 22, 25, 34, 64, 63]

Without optimization: 6 passes (always)
With optimization: 2 passes (stops when no swaps)

For already sorted array: [11, 12, 22, 25, 34, 64, 90]
Without: O(n²)
With: O(n) - only 1 pass!
```

Bubble Sort with Visualization:

```
void bubbleSortVisualize(int arr[], int n) {
    printf("\n=== Bubble Sort Visualization ===\n");

    for (int i = 0; i < n - 1; i++) {
        printf("\nPass %d:\n", i + 1);
        int swapped = 0;
```

```
        for (int j = 0; j < n - i - 1; j++) {
            printf("  Compare %d and %d: ", arr[j], arr[j + 1]);

            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                printf("Swap → ");
                swapped = 1;
            } else {
                printf("No swap → ");
            }

            printArray(arr, n);
        }

        if (swapped == 0) {
            printf("  Array is sorted!\n");
            break;
        }
    }
}
```

Complexity Analysis

Time Complexity:

- Best Case: O(n) - already sorted (with optimization)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$ - reverse sorted

Space Complexity: O(1) - in-place sorting

Comparisons and Swaps:

```
For array of size n:
- Comparisons: n(n-1)/2
- Swaps (worst case): n(n-1)/2

Example with n=5:
- Comparisons: 5×4/2 = 10
- Maximum swaps: 10
```

When to Use

Use Bubble Sort when:

- Educational purposes
- Very small datasets (n < 10)
- Nearly sorted data (with optimization)
- Simplicity is priority

Don't use when:

- Large datasets
- Performance is critical
- Production code

5.2 Selection Sort

Concept

Selection Sort divides the array into sorted and unsorted parts, repeatedly selects the minimum element from unsorted part and places it at the beginning.

How it works:

1. Find minimum element in unsorted part
2. Swap with first element of unsorted part
3. Move boundary of sorted part
4. Repeat until array is sorted

Visual Example:

```
Initial: [64, 25, 12, 22, 11]
         ↑ unsorted part

Pass 1: Find minimum (11)
[64, 25, 12, 22, 11]
                 ↑ minimum
[11, 25, 12, 22, 64]  → swap 11 and 64
 ↑   ↑ unsorted part
sorted

Pass 2: Find minimum (12)
[11, 25, 12, 22, 64]
         ↑ minimum
```

```
[11, 12, 25, 22, 64]  → swap 12 and 25
 ↑   ↑   ↑ unsorted
   sorted

Pass 3: Find minimum (22)
[11, 12, 25, 22, 64]
             ↑ minimum
[11, 12, 22, 25, 64]  → swap 22 and 25

Pass 4: Find minimum (25)
[11, 12, 22, 25, 64]
             ↑ already minimum
No swap needed

Final: [11, 12, 22, 25, 64]
```

Implementation

```
#include <stdio.h>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Find minimum element in unsorted part
        int minIndex = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        // Swap minimum with first element of unsorted part
        if (minIndex != i) {
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }
}
```

```
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Selection Sort with Visualization:

```
void selectionSortVisualize(int arr[], int n) {
    printf("\n=== Selection Sort Visualization ===\n");

    for (int i = 0; i < n - 1; i++) {
        printf("\nPass %d: ", i + 1);

        // Print sorted part
        printf("Sorted[");
        for (int k = 0; k < i; k++) {
            printf("%d", arr[k]);
            if (k < i - 1) printf(", ");
        }
        printf("] ");

        int minIndex = i;
        printf("Finding minimum in unsorted part...\n");
```

```
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        printf("  Minimum: %d at index %d\n", arr[minIndex], minIndex);

        if (minIndex != i) {
            printf("  Swapping %d and %d\n", arr[i], arr[minIndex]);
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        } else {
            printf("  Already in correct position\n");
        }

        printf("  Result: ");
        printArray(arr, n);
    }
}
```

Finding Maximum (Descending Order):

```
void selectionSortDescending(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Find maximum element in unsorted part
        int maxIndex = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] > arr[maxIndex]) {  // Changed to >
                maxIndex = j;
            }
        }

        // Swap
        if (maxIndex != i) {
            int temp = arr[i];
            arr[i] = arr[maxIndex];
```

```
            arr[maxIndex] = temp;
        }
    }
}
```

Complexity Analysis

Time Complexity:

- Best Case: O(n²)
- Average Case: O(n²)
- Worst Case: O(n²)
- **Always O(n²)** regardless of input!

Space Complexity: O(1) - in-place sorting

Comparisons and Swaps:

```
For array of size n:
- Comparisons: n(n-1)/2 (always)
- Swaps: O(n) - maximum n-1 swaps

Advantage: Minimum number of swaps!
```

Comparison with Bubble Sort:

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

Bubble Sort:
- Comparisons: 10
- Swaps: 10

Selection Sort:
- Comparisons: 10
- Swaps: 2 (only swap 5↔1, then 4↔2)

→ Selection Sort better when swaps are expensive!
```

Stability Issue

Selection Sort is **NOT stable**:

```
Input:  [4a, 3, 4b, 2]
Output: [2, 3, 4b, 4a]  ← Order of 4a and 4b changed!


Why? When we swap 4a with 2, 4b comes before 4a
```

When to Use

Use Selection Sort when:

- Swaps are expensive (writing to flash memory)
- Small datasets
- Memory writes must be minimized
- Simplicity needed

Don't use when:

- Stability required
- Large datasets
- Performance critical

5.3 Insertion Sort

Concept

Insertion Sort builds the final sorted array one item at a time by inserting each element into its correct position.

Analogy: Like sorting playing cards in your hand:

- Pick one card at a time
- Insert it into correct position among sorted cards
- Shift other cards to make space

How it works:

1. Start with second element
2. Compare with elements in sorted part (left side)
3. Shift larger elements to the right
4. Insert element in correct position
5. Repeat for all elements

Visual Example:

```
Initial: [12, 11, 13, 5, 6]
```

```
Pass 1: Insert 11
[12, 11, 13, 5, 6]
 ↑   ↑
sorted | to insert

12 > 11, shift 12 right
[11, 12, 13, 5, 6]
 ↑   ↑
   sorted

Pass 2: Insert 13
[11, 12, 13, 5, 6]
 ↑   ↑   ↑
   sorted | to insert

13 > 12, already in position
[11, 12, 13, 5, 6]
 ↑   ↑   ↑
    sorted

Pass 3: Insert 5
[11, 12, 13, 5, 6]
 ↑   ↑   ↑   ↑
    sorted  | to insert

13 > 5, shift right → [11, 12, 13, 13, 6]
12 > 5, shift right → [11, 12, 12, 13, 6]
11 > 5, shift right → [11, 11, 12, 13, 6]
Insert 5 at position 0 → [5, 11, 12, 13, 6]

Pass 4: Insert 6
[5, 11, 12, 13, 6]
 ↑  ↑   ↑   ↑   ↑
      sorted   | to insert

13 > 6, shift right → [5, 11, 12, 13, 13]
12 > 6, shift right → [5, 11, 12, 12, 13]
11 > 6, shift right → [5, 11, 11, 12, 13]
5 < 6, insert at position 1 → [5, 6, 11, 12, 13]
```

```
Final: [5, 6, 11, 12, 13]
```

Implementation

Basic Insertion Sort:

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];  // Element to be inserted
        int j = i - 1;

        // Move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        // Insert key at correct position
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    insertionSort(arr, n);
```

```
    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Insertion Sort with Visualization:

```
void insertionSortVisualize(int arr[], int n) {
    printf("\n=== Insertion Sort Visualization ===\n");
    printf("Initial: ");
    printArray(arr, n);

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

        printf("\nPass %d: Insert %d\n", i, key);
        printf("  Sorted part: [");
        for (int k = 0; k < i; k++) {
            printf("%d", arr[k]);
            if (k < i - 1) printf(", ");
        }
        printf("]\n");

        int shifts = 0;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
            shifts++;
        }

        arr[j + 1] = key;

        printf("  Shifts: %d\n", shifts);
        printf("  Result: ");
        printArray(arr, n);
    }
}
```

Descending Order:

```
void insertionSortDescending(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Change condition to arr[j] < key
        while (j >= 0 && arr[j] < key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}
```

Binary Insertion Sort (Optimization):

```
// Find position using binary search
int binarySearch(int arr[], int item, int low, int high) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == item) {
            return mid + 1;
        }
        else if (arr[mid] < item) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }

    return low;
}

void binaryInsertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
```

```
        int key = arr[i];

        // Find position using binary search
        int pos = binarySearch(arr, key, 0, i - 1);

        // Shift elements
        for (int j = i - 1; j >= pos; j--) {
            arr[j + 1] = arr[j];
        }

        // Insert key
        arr[pos] = key;
    }
}
```

Insertion Sort for Linked List:

```
struct Node {
    int data;
    struct Node *next;
};

void sortedInsert(struct Node **head, struct Node *newNode) {
    // If list is empty or new node should be first
    if (*head == NULL || (*head)->data >= newNode->data) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    // Find position to insert
    struct Node *current = *head;
    while (current->next != NULL && current->next->data < newNode->data) {
        current = current->next;
    }

    newNode->next = current->next;
    current->next = newNode;
}
```

```
void insertionSortList(struct Node **head) {
    struct Node *sorted = NULL;
    struct Node *current = *head;

    while (current != NULL) {
        struct Node *next = current->next;
        sortedInsert(&sorted, current);
        current = next;
    }

    *head = sorted;
}
```

Complexity Analysis

Time Complexity:

- Best Case: O(n) - already sorted
- Average Case: O(n²)
- Worst Case: O(n²) - reverse sorted

Space Complexity: O(1) - in-place sorting

Detailed Analysis:

```
Best Case (sorted): [1, 2, 3, 4, 5]
- Comparisons: n-1 (each element compared once)
- Shifts: 0
- Time: O(n)

Worst Case (reverse sorted): [5, 4, 3, 2, 1]
- Comparisons: 1+2+3+...+(n-1) = n(n-1)/2
- Shifts: Same as comparisons
- Time: O(n²)

Average Case (random):
- Comparisons: ~n²/4
- Shifts: ~n²/4
- Time: O(n²)
```

Performance on Different Inputs:

```
void testInsertionSort() {
    // Already sorted
    int sorted[] = {1, 2, 3, 4, 5};
    printf("Sorted array: Fast! O(n)\n");

    // Reverse sorted
    int reverse[] = {5, 4, 3, 2, 1};
    printf("Reverse array: Slow! O(n²)\n");

    // Nearly sorted
    int nearlySorted[] = {1, 2, 4, 3, 5};
    printf("Nearly sorted: Fast! O(n)\n");

    // Few elements out of place
    int fewMoves[] = {2, 1, 3, 4, 5};
    printf("Few out of place: Fast!\n");
}
```

Advantages

1. **Simple implementation**
2. **Stable** - preserves relative order
3. **In-place** - O(1) extra space
4. **Adaptive** - O(n) for nearly sorted data
5. **Online** - can sort as data arrives

Stability Example:

```
Input:  [4a, 3, 4b, 2]
Output: [2, 3, 4a, 4b]  ← Order preserved!
```

When to Use

Use Insertion Sort when:

- Small datasets (n < 50)
- Nearly sorted data
- Stability required
- Online sorting (data arriving in real-time)
- Linked lists (no shifting overhead)

Don't use when:

- Large datasets
- Random data
- Performance critical

Real-World Applications:

1. Sorting small files
2. Hybrid sorting (used in TimSort)
3. Online algorithms
4. Sorting linked lists

Revision #2
Created 2025-11-03 02:03:58 UTC by DS
Updated 2025-11-03 02:05:52 UTC by DS