Sorting Algorithms

Sorting Algorithms: Bubble Sort, Selection Sort, and Insertion Sort

1. Bubble Sort

Algorithm Explanation

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. The algorithm gets its name because smaller elements "bubble" to the beginning of the list, just like air bubbles rise to the surface of water.

How it Works:

1. Compare the first two elements
2. If the first is greater than the second, swap them
3. Move to the next pair and repeat
4. Continue until the end of the array
5. Repeat the entire process until no swaps are needed

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

Example Walkthrough:

Initial Array: [64, 34, 25, 12, 22]

Pass 1:

- Compare 64 & 34 → Swap → [34, 64, 25, 12, 22]
- Compare 64 & 25 → Swap → [34, 25, 64, 12, 22]
- Compare 64 & 12 → Swap → [34, 25, 12, 64, 22]
- Compare 64 & 22 → Swap → [34, 25, 12, 22, 64]

Pass 2:

- Compare 34 & 25 → Swap → [25, 34, 12, 22, 64]
- Compare 34 & 12 → Swap → [25, 12, 34, 22, 64]
- Compare 34 & 22 → Swap → [25, 12, 22, 34, 64]

Pass 3:

- Compare 25 & 12 → Swap → [12, 25, 22, 34, 64]
- Compare 25 & 22 → Swap → [12, 22, 25, 34, 64]

Pass 4:

- Compare 12 & 22 → No swap needed
- **Final Result:** [12, 22, 25, 34, 64]

Why Bubble Sort Takes (n-1) Passes?

Key Insight: After each complete pass through the array, the largest unsorted element is guaranteed to be in its correct final position.

- **Pass 1:** The largest element bubbles to the last position
- **Pass 2:** The second largest element bubbles to the second-last position
- **Pass 3:** The third largest element bubbles to the third-last position
- ...and so on

Since after (n-1) passes, the first (n-1) largest elements are in their correct positions, the remaining single element (the smallest) is automatically in its correct position. Therefore, we only need (n-1) passes to sort n elements.

Pseudocode:

```
for i = 0 to n-1:
    for j = 0 to n-2:
        if arr[j] > arr[j+1]:
            swap(arr[j], arr[j+1])
```

C++ Implementation:

```
#include <iostream>
#include <vector>
using namespace std;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false; // Optimization flag
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // If no swapping occurred, array is sorted
        if (!swapped) break;
    }
}

// Example usage
int main() {
    vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
    cout << "Original array: ";
    for (int x : arr) cout << x << " ";

    bubbleSort(arr);

    cout << "\nSorted array: ";
    for (int x : arr) cout << x << " ";
    return 0;
}
```

2. Selection Sort

Algorithm Explanation

Selection Sort works by finding the minimum element from the unsorted portion and placing it at the beginning. It divides the array into two parts: sorted and unsorted. Initially, the sorted part is empty and the unsorted part contains all elements.

How it Works:

1. Find the minimum element in the entire array
2. Swap it with the first element
3. Find the minimum element in the remaining unsorted array
4. Swap it with the second element
5. Continue until the entire array is sorted

Time Complexity: O(n²)

Space Complexity: O(1)

Example Walkthrough:

Initial Array: [64, 25, 12, 22, 11]

Pass 1: Find minimum (11), swap with first element

- [11, 25, 12, 22, 64]

Pass 2: Find minimum in [25, 12, 22, 64] which is 12, swap with 25

- [11, 12, 25, 22, 64]

Pass 3: Find minimum in [25, 22, 64] which is 22, swap with 25

- [11, 12, 22, 25, 64]

Pass 4: Find minimum in [25, 64] which is 25, no swap needed

- [11, 12, 22, 25, 64]

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

Why Selection Sort Takes (n-1) Passes?

Key Insight: In each pass, we select the minimum element from the unsorted portion and place it in its correct position.

- **Pass 1:** Select minimum from entire array, place at position 0
- **Pass 2:** Select minimum from positions 1 to n-1, place at position 1
- **Pass 3:** Select minimum from positions 2 to n-1, place at position 2
- ...and so on

After (n-1) passes, we have placed the first (n-1) smallest elements in their correct positions. The last remaining element is automatically the largest and is already in its correct position. Therefore, we need exactly (n-1) passes.

Pseudocode:

```
for i = 0 to n-2:
    min_index = i
    for j = i+1 to n-1:
        if arr[j] < arr[min_index]:
            min_index = j
    swap(arr[i], arr[min_index])
```

C++ Implementation:

```
#include <iostream>
#include <vector>
using namespace std;

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

    for (int i = 0; i < n - 1; i++) {
        int min_idx = i; // Find the minimum element in remaining array

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

        // Swap the found minimum element with the first element
        if (min_idx != i) {
            swap(arr[min_idx], arr[i]);
        }
    }
}

// Example usage
int main() {
    vector<int> arr = {64, 25, 12, 22, 11};
    cout << "Original array: ";
    for (int x : arr) cout << x << " ";

    selectionSort(arr);

    cout << "\nSorted array: ";
    for (int x : arr) cout << x << " ";
    return 0;
}
```

3. Insertion Sort

Algorithm Explanation

Insertion Sort builds the final sorted array one element at a time. It's similar to how you might sort playing cards in your hands - you pick up cards one by one and insert each into its proper position among the cards you've already sorted.

How it Works:

1. Start with the second element (index 1)
2. Compare it with elements before it
3. Shift larger elements to the right
4. Insert the current element in its correct position
5. Move to the next element and repeat

Time Complexity: O(n^2) worst case, O(n) best case

Space Complexity: O(1)

Example Walkthrough:

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

Pass 1: Insert 11 into sorted portion [12]

- Compare 11 with 12, shift 12 right, insert 11
- [11, 12, 13, 5, 6]

Pass 2: Insert 13 into sorted portion [11, 12]

- 13 > 12, so no shifting needed
- [11, 12, 13, 5, 6]

Pass 3: Insert 5 into sorted portion [11, 12, 13]

- Shift 13, 12, 11 to right, insert 5 at beginning
- [5, 11, 12, 13, 6]

Pass 4: Insert 6 into sorted portion [5, 11, 12, 13]

- Shift 13, 12, 11 to right, insert 6 after 5
- [5, 6, 11, 12, 13]

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

Why Insertion Sort Takes (n-1) Passes?

Key Insight: We start with the assumption that the first element is already "sorted" (a single element is trivially sorted). Then we insert each subsequent element into its correct position within the already sorted portion.

- **Pass 1:** Insert element at index 1 into sorted portion [0]
- **Pass 2:** Insert element at index 2 into sorted portion [0,1]
- **Pass 3:** Insert element at index 3 into sorted portion [0,1,2]
- ...and so on

After (n-1) passes, we have processed all elements from index 1 to n-1, inserting each into the sorted portion. Since we started with element 0 being "sorted", we need exactly (n-1) insertions to sort n elements.

Pseudocode:

```
for i = 1 to n-1:
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key:
        arr[j+1] = arr[j]
        j = j - 1
    arr[j+1] = key
```

C++ Implementation:

```cpp
#include <iostream>
#include <vector>
using namespace std;

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

    for (int i = 1; i < n; i++) {
        int key = arr[i]; // Current 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 = j - 1;
        }
        // Insert key at its correct position
        arr[j + 1] = key;
    }
}

// Example usage
int main() {
    vector<int> arr = {12, 11, 13, 5, 6};
    cout << "Original array: ";
    for (int x : arr) cout << x << " ";

    insertionSort(arr);

    cout << "\nSorted array: ";
    for (int x : arr) cout << x << " ";
    return 0;
}
```

Key Differences Summary

Algorithm	Best Case	Average Case	Worst Case	Space	Stability	When to Use
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Stable	Educational purposes, very small datasets
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Unstable	When memory writes are costly
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Stable	Small datasets, nearly sorted data

Stable: Maintains relative order of equal elements
Unstable: May change relative order of equal elements

Problems to practice

Problem 1: Bubble Sort with Swap Count

Problem Statement:
Consider the following version of Bubble Sort and print:

1. Number of swaps it took to sort the array
2. First element in the array after sorting
3. Last element in the array after sorting

Input Format:

- First line: integer n (size of array)

- Second line: n space-separated integers

Output Format:
Print the three required values on separate lines.

Example 1:

```
Input:
3
6 4 1

Output:
3
1
6
```

Example 2:

```
Input:
3
1 2 3

Output:
0
1
3
```

Solution Approach:

1. Implement bubble sort while counting swaps
2. Track the number of swaps made
3. After sorting, output the swap count, first, and last elements

---

Problem 2: Index of Target Element After Sorting

Problem Statement:
Find the index of a target element after sorting the array (0-based indexing).

Input Format:

- First line: integer n (size of array)
- Second line: n space-separated integers
- Third line: target element

Output Format:
Index of target element after sorting

Example 1:

```
Input:
6
```

```
6 27 2 3 1 5
5

Output:
3
```

Explanation: Sorted array = [1, 2, 3, 5, 6, 27], index of 5 is 3

Example 2:

```
Input:
7
9 7 19 20 13 1 6
6

Output:
1
```

Explanation: Sorted array = [1, 6, 7, 9, 13, 19, 20], index of 6 is 1

Problem 3: Find Smallest Greater Elements (Using Bubble Sort)

Problem Statement:
For each element in the array, find the smallest element that is greater than it. If no such element exists, use -10000000.

Input Format:

- First line: integer n (size of array)
- Second line: n space-separated integers

Output Format:
Array of n elements containing smallest greater elements

Example 1:

```
Input:
4
13 6 17 12

Output:
17 12 -10000000 13
```

Example 2:

```
Input:
9
6 3 9 8 10 2 1 15 7

Output:
7 6 10 9 15 3 2 -10000000 8
```

Solution Approach:

1. For each element, find all elements greater than it
2. Among those, find the minimum
3. If no greater element exists, use -10000000

Problem 4: Column-wise Matrix Sorting

Problem Statement:
Sort each column of a matrix in ascending order.

Input Format:

- First line: integers N and M (rows and columns)
- Next N lines: M integers each (matrix elements)

Output Format:
Matrix after sorting each column

Example 1:

```
Input:
3 5
9 7 8 11 21
1 4 3 7 2
4 3 14 9 12

Output:
1 3 3 7 2
4 4 8 9 12
9 7 14 11 21
```

Example 2:

```
Input:
5 9
9 14 62 23 25 25 41 33 95
78 7 30 97 51 35 41 42 92
79 32 45 30 62 92 87 8 19
52 100 36 11 57 85 73 91 54
90 94 98 21 12 79 80 78 72

Output:
9 7 30 11 12 25 41 8 19
52 14 36 21 25 35 41 33 54
78 32 45 23 51 79 73 42 72
79 94 62 30 57 85 80 78 92
90 100 98 97 62 92 87 91 95
```

Solution Approach:

1. For each column, extract all elements
2. Sort the column using any sorting algorithm
3. Place sorted elements back in the matrix
4. Print the final matrix

LeetCode Problems with Solutions

Problem 1: Sort an Array (LeetCode #912)

Link: https://leetcode.com/problems/sort-an-array/

Problem Statement:
Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions.

Example:

```
Input: nums = [5,2,3,1]
Output: [1,2,3,5]
```

Solution 1 - Using Bubble Sort:

```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n - 1; i++) {
            bool swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (nums[j] > nums[j + 1]) {
                    swap(nums[j], nums[j + 1]);
                    swapped = true;
                }
            }
            if (!swapped) break; // Array is already sorted
        }
        return nums;
    }
};
```

Solution 2 - Using Selection Sort:

```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n - 1; i++) {
            int min_idx = i;
            for (int j = i + 1; j < n; j++) {
                if (nums[j] < nums[min_idx]) {
                    min_idx = j;
                }
            }
            if (min_idx != i) {
                swap(nums[min_idx], nums[i]);
            }
        }
        return nums;
    }
};
```

Solution 3 - Using Insertion Sort:

```
class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        int n = nums.size();
        for (int i = 1; i < n; i++) {
            int key = nums[i];
            int j = i - 1;
            while (j >= 0 && nums[j] > key) {
                nums[j + 1] = nums[j];
                j--;
            }
            nums[j + 1] = key;
        }
        return nums;
    }
};
```

Problem 2: Sort Colors (LeetCode #75)

Link: https://leetcode.com/problems/sort-colors/

Problem Statement:
Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue. We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

Example:

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Solution - Using Bubble Sort Approach:

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int n = nums.size();
        for (int i = 0; i < n - 1; i++) {
            bool swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (nums[j] > nums[j + 1]) {
                    swap(nums[j], nums[j + 1]);
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
    }
};
```

Problem 3: Merge Sorted Array (LeetCode #88)

Link: https://leetcode.com/problems/merge-sorted-array/

Problem Statement:
You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively. Merge nums1 and nums2 into a single array sorted in non-decreasing order.

Example:

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
```

Solution - Merge and Sort using Insertion Sort:

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        // Copy nums2 to the end of nums1
        for (int i = 0; i < n; i++) {
            nums1[m + i] = nums2[i];
        }

        // Apply insertion sort to the merged array
        int size = m + n;
        for (int i = 1; i < size; i++) {
            int key = nums1[i];
            int j = i - 1;
            while (j >= 0 && nums1[j] > key) {
                nums1[j + 1] = nums1[j];
                j--;
            }
            nums1[j + 1] = key;
        }
    }
};
```

Problem 4: Squares of a Sorted Array (LeetCode #977)

Link: https://leetcode.com/problems/squares-of-a-sorted-array/

Problem Statement:
Given an integer array nums sorted in non-decreasing order, return an array of the squares of each number sorted in non-decreasing order.

Example:

```
Input: nums = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
```

Solution - Square and Sort using Selection Sort:

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        int n = nums.size();

        // Square all elements
        for (int i = 0; i < n; i++) {
            nums[i] = nums[i] * nums[i];
        }

        // Apply selection sort
        for (int i = 0; i < n - 1; i++) {
            int min_idx = i;
            for (int j = i + 1; j < n; j++) {
                if (nums[j] < nums[min_idx]) {
                    min_idx = j;
                }
            }
            if (min_idx != i) {
                swap(nums[min_idx], nums[i]);
            }
        }

        return nums;
    }
};
```

Problem 5: Sort List (LeetCode #148)

Link: https://leetcode.com/problems/sort-list/

Problem Statement:
Given the head of a linked list, return the list after sorting it in ascending order.

Example:

```
Input: head = [4,2,1,3]
Output: [1,2,3,4]
```

Solution - Convert to Array, Sort using Insertion Sort, Rebuild List:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;

        // Convert linked list to vector
        vector<int> values;
        ListNode* curr = head;
        while (curr) {
            values.push_back(curr->val);
            curr = curr->next;
        }

        // Sort using insertion sort
        int n = values.size();
        for (int i = 1; i < n; i++) {
            int key = values[i];
            int j = i - 1;
            while (j >= 0 && values[j] > key) {
                values[j + 1] = values[j];
                j--;
            }
            values[j + 1] = key;
        }

        // Rebuild linked list
        curr = head;
        for (int val : values) {
            curr→val = val;
            curr = curr→next;
        }

        return head;
    }
};
```

Additional Practice Problems