Bubble Sort is another simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

How Bubble Sort Works

1. **Compare Adjacent Elements:**
    - Start at the beginning of the array and compare the first two adjacent elements.
    - If the first element is greater than the second element, swap them.
2. **Continue for the Whole Array:**
    - Move to the next pair of adjacent elements and repeat the comparison and swap if needed.
    - Continue this process for the entire array. After the first pass, the largest element will have "bubbled up" to the end of the array.
3. **Repeat:**
    - Repeat the process for the remaining unsorted portion of the array (i.e., all elements except the last one which is already in its correct place).
    - Each subsequent pass will have one less element to check because the end of the array is sorted.
4. **Termination:**
    - Continue this process until no swaps are needed in a pass, indicating that the array is sorted.

Complexity

- **Time Complexity:** (O(n^2)) due to the nested loops. It can be (O(n)) if the array is already sorted (with an optimized version).
- **Space Complexity:** (O(1)) because it sorts the array in place.

Example in C++

Here's a C++ implementation of Bubble Sort:

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

// Function to perform bubble sort
void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;  // Reset the swap flag for each pass
        for (int j = 0; j < n - i - 1; j++) {
            // Compare adjacent elements
            if (arr[j] > arr[j + 1]) {
                // Swap if they are in the wrong order
                int temp = arr[j];
                arr[j] = arr[j + 1];
```

```
                arr[j + 1] = temp;
                swapped = true;  // Set the swap flag
            }
        }
        // If no elements were swapped, the array is already sorted
        if (!swapped) {
            break;
        }
    }
}

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

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

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

    bubbleSort(arr, n);

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

    return 0;
}
```

Explanation of the Code

1. `bubbleSort` **Function:**
   - **Outer Loop:** Repeats for each element except the last one.
   - **Inner Loop:** Compares each pair of adjacent elements and swaps them if needed.
   - **Swapped Flag:** Checks if any swaps were made in the current pass. If no swaps occurred, the array is sorted, and the loop can be terminated early.
2. `printArray` **Function:**
   - Prints the elements of the array to the console.
3. `main` **Function:**
   - Initializes an array, prints it, sorts it using `bubbleSort`, and then prints the sorted array.

Bubble Sort is easy to understand and implement, but it is not very efficient for large lists compared to more advanced sorting algorithms like Quick Sort or Merge Sort.

If you change the outer loop in your `bubbleSort` implementation to start from `i = 0` and run until `i < n - 1`, you need to adjust the inner loop accordingly. This change affects how you iterate over the elements and how many passes you need to make.

Here's the modified code to reflect this change:

```
#include <bits/stdc++.h>
using namespace std;

void bubbleSort(vector<int>& arr, int n) {
    for(int i = 0; i < n - 1; i++) {
        // Iterate through the array up to the n-i-1 element
        for(int j = 0; j < n - i - 1; j++) {
            if(arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void printArray(const vector<int>& arr) {
    for(int num : arr) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {64, 25, 12, 22, 11};
    int n = arr.size();

    cout << "Unsorted array: ";
    printArray(arr);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr);

    return 0;
}
```

Explanation

1. **Outer Loop (`i` Loop):**
    - **Starts from `0`:** This loop iterates from `0` to `n - 2` (total `n - 1` passes), which ensures that the algorithm performs enough passes to sort the array.

2. **Inner Loop (`j` Loop):**
    - **Runs from `0` to `n - i - 2`:** On each pass of the outer loop, the largest element among the unsorted portion is moved to its correct position. The inner loop decreases its range each pass because the last elements are sorted.

Why `n - i - 1`?

- In the first pass, you need to check `n - 1` pairs of adjacent elements.
- In the second pass, you only need to check up to `n - 2` pairs, as the last element is already sorted.
- This continues until the loop covers fewer and fewer elements, hence the `n - i - 1` range.

By adjusting the loops this way, the code ensures that the Bubble Sort algorithm runs correctly and efficiently, minimizing unnecessary comparisons as the list becomes increasingly sorted.

Certainly! Let's break down the Bubble Sort algorithm step-by-step with all iterations and a sample array to make it clear. We'll use the example array `[5, 3, 8, 4, 2]` and explain each iteration in detail.

Bubble Sort Algorithm

Here's the Bubble Sort function for reference:

```
void bubbleSort(vector<int>& arr, int n) {
    for (int i = 0; i < n - 1; i++) {         // Outer loop
        for (int j = 0; j < n - i - 1; j++) { // Inner loop
            if (arr[j] > arr[j + 1]) {        // Compare adjacent elements
                swap(arr[j], arr[j + 1]);     // Swap if needed
            }
        }
    }
}
```

Array: `[5, 3, 8, 4, 2]`

Initial Array:

```
[5, 3, 8, 4, 2]
```

Pass-by-Pass Breakdown

Pass 1 (i = 0)

- **Iteration 1 (j = 0):** Compare `5` and `3`. Since `5 > 3`, swap them.

  ```
  [3, 5, 8, 4, 2]
  ```

- **Iteration 2 (j = 1):** Compare 5 and 8. No swap needed.

```
[3, 5, 8, 4, 2]
```

- **Iteration 3 (j = 2):** Compare 8 and 4. Since 8 > 4, swap them.

```
[3, 5, 4, 8, 2]
```

- **Iteration 4 (j = 3):** Compare 8 and 2. Since 8 > 2, swap them.

```
[3, 5, 4, 2, 8]
```

After Pass 1:

```
[3, 5, 4, 2, 8]
```

The largest element 8 is now in its correct position at the end.

Pass 2 (i = 1)

- **Iteration 1 (j = 0):** Compare 3 and 5. No swap needed.

```
[3, 5, 4, 2, 8]
```

- **Iteration 2 (j = 1):** Compare 5 and 4. Since 5 > 4, swap them.

```
[3, 4, 5, 2, 8]
```

- **Iteration 3 (j = 2):** Compare 5 and 2. Since 5 > 2, swap them.

```
[3, 4, 2, 5, 8]
```

After Pass 2:

```
[3, 4, 2, 5, 8]
```

The second largest element 5 is now in its correct position.

Pass 3 (i = 2)

- **Iteration 1 (j = 0):** Compare 3 and 4. No swap needed.

```
[3, 4, 2, 5, 8]
```

- **Iteration 2 (j = 1):** Compare 4 and 2. Since 4 > 2, swap them.

```
[3, 2, 4, 5, 8]
```

After Pass 3:

```
[3, 2, 4, 5, 8]
```

The third largest element 4 is now in its correct position.

Pass 4 (i = 3)

- **Iteration 1 (j = 0):** Compare 3 and 2. Since 3 > 2, swap them.

```
[2, 3, 4, 5, 8]
```

After Pass 4:

```
[2, 3, 4, 5, 8]
```

The remaining elements are already sorted.

Summary of Passes

1. **Pass 1:** The largest element 8 is moved to the end.
2. **Pass 2:** The second largest element 5 is placed in the correct position.
3. **Pass 3:** The third largest element 4 is correctly positioned.
4. **Pass 4:** The remaining elements are sorted.

Final Sorted Array

After all passes, the array is:

```
[2, 3, 4, 5, 8]
```

Key Points

- **Outer Loop (i Loop):** Runs from `0` to `n-2` because each pass places the largest unsorted element in its correct position.
- **Inner Loop (j Loop):** Runs from `0` to `n-i-2` to avoid redundant comparisons with already sorted elements.
- **Swapping:** Ensures that larger elements "bubble up" to the end of the array.

This step-by-step explanation illustrates how Bubble Sort works by progressively moving the largest unsorted elements to their correct positions.