Bubble sort in C++

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

void BubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int arr[] = {0, 1, 5, 7, 8, 9, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    BubbleSort(arr, n);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Dry Run Table:

Initial:

[0, 1, 5, 7, 8, 9, 4]

Pass 1 (i = 0):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 7, 8, 9, 4]
1 and 5	No	[0, 1, 5, 7, 8, 9, 4]
5 and 7	No	[0, 1, 5, 7, 8, 9, 4]
7 and 8	No	[0, 1, 5, 7, 8, 9, 4]
8 and 9	No	[0, 1, 5, 7, 8, 9, 4]
9 and 4	**Yes**	[0, 1, 5, 7, 8, 4, 9]

✅ Largest element 9 moved to the end.

Pass 2 (i = 1):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 7, 8, 4, 9]
1 and 5	No	[0, 1, 5, 7, 8, 4, 9]
5 and 7	No	[0, 1, 5, 7, 8, 4, 9]
7 and 8	No	[0, 1, 5, 7, 8, 4, 9]
8 and 4	**Yes**	[0, 1, 5, 7, 4, 8, 9]

✅ Second-largest 8 in place.

Pass 3 (i = 2):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 7, 4, 8, 9]
1 and 5	No	[0, 1, 5, 7, 4, 8, 9]
5 and 7	No	[0, 1, 5, 7, 4, 8, 9]
7 and 4	**Yes**	[0, 1, 5, 4, 7, 8, 9]

Pass 4 (i = 3):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 5, 4, 7, 8, 9]
1 and 5	No	[0, 1, 5, 4, 7, 8, 9]
5 and 4	**Yes**	[0, 1, 4, 5, 7, 8, 9]

Pass 5 (i = 4):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 4, 5, 7, 8, 9]

Compare arr[j]	Swap?	Result
1 and 4	No	[0, 1, 4, 5, 7, 8, 9]

Pass 6 (i = 5):

Compare arr[j]	Swap?	Result
0 and 1	No	[0, 1, 4, 5, 7, 8, 9]

🏁 **Final Sorted Array:**

Sorted array: 0 1 4 5 7 8 9

Sorted array: 0 1 4 5 7 8 9

Count Inversions in C++

```
#include <iostream>
#include <vector>

using namespace std;

long long ans;

void merge(vector<long long>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    vector<long long> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
            ans += (m - l + 1 - i);
        }
    }

    while (i < n1) {
        arr[k++] = L[i++];
    }

    while (j < n2) {
        arr[k++] = R[j++];
    }
}

void mergeSort(vector<long long>& arr, int l, int r)
{
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

long long inversionCount(vector<long long>& arr) {
    ans = 0;
    mergeSort(arr, 0, arr.size() - 1);
    return ans;
}

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

Step-by-Step Merge and Inversion Tracking

Step	Subarrays (Left - Right)	Comparison	Inversion Count	Merged Result
1	[2] and [3]	2 ≤ 3	0	[2, 3]
2	[2, 3] and [8]	All in order	0	[2, 3, 8]
3	[6] and [1]	6 > 1	1	[1, 6]
4	[2, 3, 8] and [1, 6]	2 > 1	3 (2,3,8 > 1)	
		2 < 6	0	
		3 < 6	0	
		8 > 6	1	[1, 2, 3, 6, 8]

✅ Summary

Merge Step	Inversions Found
[2] and [3]	0
[2, 3] and [8]	0
[6] and [1]	1
[2, 3, 8] and [1, 6]	3 + 1 = 4
Total Inversions	**5**

```
int main() {
    vector<long long> arr = {2, 3, 8, 6, 1};

    cout << "Given Array:" << endl;
    printArray(arr);

    long long inversionCountValue =
inversionCount(arr);

    cout << "Number of inversions: " <<
inversionCountValue << endl;

    return 0;
}
```

```
Given Array:
2 3 8 6 1
Number of inversions: 5
```

Count Sort in C++

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

string countSort(string s) {
    char arr[s.length()];
    strcpy(arr, s.c_str());

    char maxch = 'a';
    for (int i = 0; i < strlen(arr); i++) {
        if (arr[i] > maxch) {
            maxch = arr[i];
        }
    }
    int max = maxch - 'a';
    int count[max + 1] = {0};

    for (int i = 0; i < strlen(arr); i++) {
        int val = arr[i] - 'a';
        count[val]++;
    }

    int k = 0;
    for (int i = 0; i <= max; i++) {
        int c = count[i];
        for (int j = 0; j < c; j++) {
            arr[k] = i + 'a';
            k++;
        }
    }

    string sortedString(arr);
    return sortedString;
}

int main() {
    string input = "countingsortexample";

    string sortedString = countSort(input);

    cout << "Original String: " << input << endl;
    cout << "Sorted String: " << sortedString << endl;

    return 0;
}
```

Step-by-Step Dry Run:

Step 1: Copy string to character array

```
strcpy(arr, s.c_str());
```

Now arr = "countingsortexample"

Step 2: Find max character (in terms of ASCII)

```
char maxch = 'x'; // max character = 'x'
int max = maxch - 'a'; // max = 23
```

Step 3: Count frequency of each character

Character	Count
a	1
c	1
e	2
g	1
i	1
l	1
m	1
n	2
o	2
p	1
r	1
s	1
t	2
u	1
x	1

Step 4: Reconstruct the sorted array

Characters are added in order of 'a' to 'x' based on count.

Sorted string becomes:

<table>
<tr><td></td><td>"aceegilmnnooprsttux"

✅ Output:

Original String: countingsortexample
Sorted String: aceegilmnnooprsttux</td></tr>
<tr><td colspan="2">Original String: countingsortexample
Sorted String: aceegilmnnooprsttux</td></tr>
</table>

Good Integers distinct in C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int GoodIntegers(vector<int>& arr) {
    sort(arr.begin(), arr.end()); // Sort the array

    int ans = 0;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == i) { // Check if the value at index i
matches i
            ++ans;
        }
    }

    return ans; // Return the count of good integers
}

int main() {
    vector<int> arr = {0, 1, 5, 7, 8, 9, 4};

    cout << GoodIntegers(arr) << endl;

    return 0;
}
```

Input:

vector<int> arr = {0, 1, 5, 7, 8, 9, 4};

Step 1: Sort the array

Sorted arr = {0, 1, 4, 5, 7, 8, 9}
↑ ↑ ↑ ↑ ↑ ↑ ↑
Index 0 1 2 3 4 5 6

Step 2: Compare each element with its index

Index i	arr[i]	arr[i] == i	Count (ans)
0	0	✅ Yes	1
1	1	✅ Yes	2
2	4	❌ No	2
3	5	❌ No	2
4	7	❌ No	2
5	8	❌ No	2
6	9	❌ No	2

Final Output:

cout << GoodIntegers(arr); // Output: 2

✅ Because arr[0] = 0 and arr[1] = 1 match their indices.

Good Integers duplicate in C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int GoodIntegers(int arr[], int n) {
    sort(arr, arr + n); // Sort the array

    int ans = 0;
    int lessCount = 0;

    if (arr[0] == 0) {
        ans++;
    }

    for (int i = 1; i < n; ++i) {
        if (arr[i] != arr[i - 1]) {
            lessCount = i;
        }

        if (arr[i] == lessCount) {
            ans++;
        }
    }

    return ans;
}

int main() {
    int arr[] = {0, 1, 5, 7, 8, 9, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << GoodIntegers(arr, n) << endl;

    return 0;
}
```

Goal of the Function:

Count how many elements in the array are **equal to the number of elements less than it**.

🔄 Step-by-step Dry Run

➤ Step 1: Sort the array

Initial array: {0, 1, 5, 7, 8, 9, 4}
Sorted array: {0, 1, 4, 5, 7, 8, 9}
n = 7
Variables: ans = 0, lessCount = 0

Index (i)	arr[i]	arr[i-1]	arr[i] != arr[i-1]	lessCount	arr[i] == lessCount	ans
0	0	-	-	0	✓ (0 == 0)	1
1	1	0	✓	1	✓ (1 == 1)	2
2	4	1	✓	2	✗ (4 != 2)	2
3	5	4	✓	3	✗ (5 != 3)	2
4	7	5	✓	4	✗ (7 != 4)	2
5	8	7	✓	5	✗ (8 != 5)	2
6	9	8	✓	6	✗ (9 != 6)	2

✓ Final Answer: 2

The two good integers are:

- 0: there are 0 elements less than it → good
- 1: there is 1 element less than it → good

Insertion Sort in C++

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

class InsertionSort {
public:
    // Function to perform insertion sort on array arr of size n
    void insertionSort(int arr[], int n) {
        for (int i = 1; i < n; i++) {
            insert(arr, i);
        }
    }

private:
    // Helper function to insert arr[i] into the sorted sub-array arr[0...i-1]
    void insert(int arr[], int i) {
        int key = arr[i]; // Element to be inserted
        int j = i - 1;    // Start comparing with the previous element

        // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key; // Place key at its correct position
    }
};

int main() {
    InsertionSort solution;

    // Hardcoded input array
    int arr[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Sorting the array using insertion sort
    solution.insertionSort(arr, n);

    // Printing the sorted array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Let's dry run your **Insertion Sort** code step by step with the input:

int arr[] = {5, 2, 9, 1, 5, 6};

🔄 Insertion Sort Dry Run Table

i	Key	Initial Array State	Comparison Index (j)	Action Taken	Updated Array
1	2	[5, 2, 9, 1, 5, 6]	j = 0 (5 > 2)	Shift 5 to index 1	[5, 5, 9, 1, 5, 6]
			j = -1	Insert 2 at index 0	[2, 5, 9, 1, 5, 6]
2	9	[2, 5, 9, 1, 5, 6]	j = 1 (5 < 9)	No shifting, insert 9 at index 2	[2, 5, 9, 1, 5, 6]
3	1	[2, 5, 9, 1, 5, 6]	j = 2 (9 > 1)	Shift 9 to index 3	[2, 5, 9, 9, 5, 6]
			j = 1 (5 > 1)	Shift 5 to index 2	[2, 5, 5, 9, 5, 6]
			j = 0 (2 > 1)	Shift 2 to index 1	[2, 2, 5, 9, 5, 6]
			j = -1	Insert 1 at index 0	[1, 2, 5, 9, 5, 6]
4	5	[1, 2, 5, 9, 5, 6]	j = 3 (9 > 5)	Shift 9 to index 4	[1, 2, 5, 9, 9, 6]
			j = 2 (5 == 5)	No shifting (stable), insert 5 at index 3	[1, 2, 5, 5, 9, 6]
5	6	[1, 2, 5, 5, 9, 6]	j = 4 (9 > 6)	Shift 9 to index 5	[1, 2, 5, 5, 9, 9]
			j = 3 (5 < 6)	Insert 6 at index 4	[1, 2, 5, 5, 6, 9]

✅ Final Sorted Array:

	[1, 2, 5, 5, 6, 9]
1 2 5 5 6 9	

Merge 2 sorted subarrays in C++

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

// Function to merge two sorted subarrays within array 'a'
vector<int> mergeTwoSortedSubArray(vector<int>& a, int s, int m, int e) {
    vector<int> temp(e - s + 1);
    int p1 = s;
    int p2 = m + 1;
    int p3 = 0;

    // Merge elements from two subarrays into temp array
    while (p1 <= m && p2 <= e) {
        if (a[p1] < a[p2]) {
            temp[p3] = a[p1];
            p3++;
            p1++;
        } else {
            temp[p3] = a[p2];
            p3++;
            p2++;
        }
    }

    // Copy remaining elements of the first subarray, if any
    while (p1 <= m) {
        temp[p3] = a[p1];
        p3++;
        p1++;
    }

    // Copy remaining elements of the second subarray, if any
    while (p2 <= e) {
        temp[p3] = a[p2];
        p3++;
        p2++;
    }

    // Copy sorted elements from temp back to original array 'a'
    for (int i = 0; i < temp.size(); i++) {
        a[s + i] = temp[i];
    }

    return a;
}

int main() {
    // Hard-coded input
    vector<int> A = {1, 3, 5, 7, 2, 4, 6, 8};
    int s = 0;
    int m = 3; // Middle index of the first sorted subarray
    int e = 7; // End index of the second sorted subarray

    // Merging the two sorted subarrays
```

using the input:

A = {1, 3, 5, 7, 2, 4, 6, 8}
s = 0, m = 3, e = 7

This means:

- First sorted subarray = A[0..3] = {1, 3, 5, 7}
- Second sorted subarray = A[4..7] = {2, 4, 6, 8}

🔄 **Dry Run Table:**

Step	p1	p2	temp[] (after step)	Comment
1	0	4	{1}	1 < 2, so copy 1 from left
2	1	4	{1, 2}	2 < 3, so copy 2 from right
3	1	5	{1, 2, 3}	3 < 4, so copy 3 from left
4	2	5	{1, 2, 3, 4}	4 < 5, so copy 4 from right
5	2	6	{1, 2, 3, 4, 5}	5 < 6, so copy 5 from left
6	3	6	{1, 2, 3, 4, 5, 6}	6 < 7, so copy 6 from right
7	3	7	{1, 2, 3, 4, 5, 6, 7}	7 < 8, so copy 7 from left
8	4	7	{1, 2, 3, 4, 5, 6, 7, 8}	only 8 left, copy from right

Now the merged array looks like:

A = {1, 2, 3, 4, 5, 6, 7, 8}

✅ **Final Output:**

Merged array: 1 2 3 4 5 6 7 8

<table>
<tr><td>

```
    vector<int> result = mergeTwoSortedSubArray(A,
s, m, e);

    // Print the result
    cout << "Merged array: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

</td><td></td></tr>
<tr><td colspan="2">Merged array: 1 2 3 4 5 6 7 8</td></tr>
</table>

Merge Sort in C++

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

class MergeSort {
public:
    void merge(vector<int>& arr, int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;

        // Create temporary arrays
        vector<int> L(n1), R(n2);

        // Copy data to temporary arrays L[] and R[]
        for (int i = 0; i < n1; i++)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; j++)
            R[j] = arr[m + 1 + j];

        // Merge the temporary arrays back into arr[l..r]
        int i = 0; // Initial index of first subarray
        int j = 0; // Initial index of second subarray
        int k = l; // Initial index of merged subarray

        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy the remaining elements of L[], if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy the remaining elements of R[], if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    void mergeSort(vector<int>& arr, int l, int r) {
        if (l >= r) {
            return; // Base case: array size is 0 or 1
        }
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);      // Sort first half
        mergeSort(arr, m + 1, r); // Sort second half
        merge(arr, l, m, r);       // Merge sorted halves
    }
};
```

Let's walk through a **dry run** of your **Merge Sort implementation** with the input:

arr = {12, 11, 13, 5, 6, 7}

🔢 Step-by-step Breakdown:

We'll visualize the recursive division and merging process.

🧩 Recursive Division (mergeSort)

Level	Call	Subarray
1	mergeSort(arr, 0, 5)	[12, 11, 13, 5, 6, 7]
2	mergeSort(arr, 0, 2)	[12, 11, 13]
3	mergeSort(arr, 0, 1)	[12, 11]
4	mergeSort(arr, 0, 0)	[12]
4	mergeSort(arr, 1, 1)	[11]
3	merge(arr, 0, 0, 1)	merge [12] and [11] ⇒ [11, 12]
3	mergeSort(arr, 2, 2)	[13]
2	merge(arr, 0, 1, 2)	merge [11, 12] and [13] ⇒ [11, 12, 13]
2	mergeSort(arr, 3, 5)	[5, 6, 7]
3	mergeSort(arr, 3, 4)	[5, 6]
4	mergeSort(arr, 3, 3)	[5]
4	mergeSort(arr, 4, 4)	[6]
3	merge(arr, 3, 3, 4)	merge [5] and [6] ⇒ [5, 6]
3	mergeSort(arr, 5, 5)	[7]
2	merge(arr, 3, 4, 5)	merge [5, 6] and [7] ⇒ [5, 6, 7]

```
int main() {
    MergeSort solution;

    // Hardcoded input array
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    int n = arr.size();

    cout << "Given Array:" << endl;
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    solution.mergeSort(arr, 0, n - 1);

    cout << "\nSorted array:" << endl;
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

1	merge(arr, 0, 2, 5)	merge [11, 12, 13] and [5, 6, 7] ⇒ [5, 6, 7, 11, 12, 13]

✅ **Final Sorted Array:**

[5, 6, 7, 11, 12, 13]

📘 **Visual of Merges**

```
Initial:     [12, 11, 13, 5, 6, 7]
Split1:      [12, 11, 13]    | [5, 6, 7]
Split2:    [12, 11] [13]     | [5, 6] [7]
Merge1:      [11, 12] + [13] = [11, 12, 13]
Merge2:      [5, 6] + [7] = [5, 6, 7]
Final Merge: [11, 12, 13] + [5, 6, 7] = [5, 6, 7, 11, 12, 13]
```

```
Given Array:
12 11 13 5 6 7

Sorted array:
5 6 7 11 12 13
```

Order of removal in C++

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

class OrderOfRemoval {
public:
    static int orderOfRemoval(vector<int>& arr) {
        int n = arr.size();
        sort(arr.begin(), arr.end()); // Sorting the array

        int ans = 0;
        for (int i = 0; i < n; i++) {
            int temp = arr[i] * (n - i);
            ans += temp;
        }

        return ans;
    }
};

int main() {
    // Hardcoded input array
    vector<int> arr = {1, 2, 3, 4, 5};
    int n = arr.size();

    // Calling orderOfRemoval function to calculate the order of removal
    int result = OrderOfRemoval::orderOfRemoval(arr);

    // Printing the result
    cout << "Order of removal: " << result << endl;

    return 0;
}
```

Let's perform a **detailed dry run** of your orderOfRemoval function using the input array:

arr = {1, 2, 3, 4, 5}

Step-by-step Dry Run:

1. **Sort the array**: The input array {1, 2, 3, 4, 5} is already sorted, so no changes are made.

   Sorted array: {1, 2, 3, 4, 5}

2. **Initialize Variables**:
   - n = arr.size() = 5
   - ans = 0 (This will hold the final result)
3. **Iterate and calculate the result**: For each element arr[i] in the array, the contribution of that element to the ans is calculated by multiplying arr[i] with the remaining elements (i.e., arr[i] * (n - i)).

Dry Run Table:

i	arr[i]	n - i	arr[i] * (n - i)	Cumulative ans
0	1	5	1 * 5 = 5	0 + 5 = 5
1	2	4	2 * 4 = 8	5 + 8 = 13
2	3	3	3 * 3 = 9	13 + 9 = 22
3	4	2	4 * 2 = 8	22 + 8 = 30
4	5	1	5 * 1 = 5	30 + 5 = 35

Final Result:

After the loop finishes, the value of ans is 35.

So, the output of the program is:

Order of removal: 35

Order of removal: 35

Subsets in C++

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

class Subsets {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        int totalno = (1 << n);
        vector<vector<int>> ans;

        for (int i = 0; i < totalno; i++) {
            vector<int> temp;
            for (int j = 0; j < n; j++) {
                if (checkBit(i, j)) {
                    temp.push_back(nums[j]);
                }
            }
            ans.push_back(temp);
        }

        return ans;
    }

private:
    // Helper function to check if the i-th bit in n is set
    bool checkBit(int n, int i) {
        return (n & (1 << i)) != 0;
    }
};

int main() {
    // Create an instance of the Subsets class
    Subsets solution;

    // Hardcoded input array
    vector<int> nums = {1, 2, 3}; // Example input

    // Calling subsets to generate all subsets of the
array
    vector<vector<int>> subsets =
solution.subsets(nums);

    // Printing all subsets
    for (auto& subset : subsets) {
        cout << "[";
        for (size_t i = 0; i < subset.size(); i++) {
            cout << subset[i];
            if (i < subset.size() - 1) {
                cout << ", ";
            }
        }
        cout << "]" << endl;
    }

    return 0;
}
```

Detailed Table:

i (Binary)	Subset Indexes	Subset Elements	Subset
0 (000)	None	None	[]
1 (001)	0	{1}	[1]
2 (010)	1	{2}	[2]
3 (011)	0, 1	{1, 2}	[1, 2]
4 (100)	2	{3}	[3]
5 (101)	0, 2	{1, 3}	[1, 3]
6 (110)	1, 2	{2, 3}	[2, 3]
7 (111)	0, 1, 2	{1, 2, 3}	[1, 2, 3]

Explanation of Each Iteration:

1. **Iteration 1 (i = 0 / Binary 000)**:
   - No bits are set, so the subset is empty: [].
2. **Iteration 2 (i = 1 / Binary 001)**:
   - Only the least significant bit is set, so the subset includes only the element 1: [1].
3. **Iteration 3 (i = 2 / Binary 010)**:
   - The second bit is set, so the subset includes only the element 2: [2].
4. **Iteration 4 (i = 3 / Binary 011)**:
   - The first and second bits are set, so the subset includes the elements 1 and 2: [1, 2].
5. **Iteration 5 (i = 4 / Binary 100)**:
   - The third bit is set, so the subset includes only the element 3: [3].
6. **Iteration 6 (i = 5 / Binary 101)**:
   - The first and third bits are set, so the subset includes the elements 1 and 3: [1, 3].
7. **Iteration 7 (i = 6 / Binary 110)**:
   - The second and third bits are set, so the subset includes the elements 2 and 3: [2, 3].
8. **Iteration 8 (i = 7 / Binary 111)**:
   - All bits are set, so the subset includes all elements: [1, 2, 3].

Final Output:

The final list of subsets is:

```
 [ ]
[1]
[2]
[1, 2]
[3]
```

<table>
<tr><td></td><td>[1, 3]
[2, 3]
[1, 2, 3]</td></tr>
<tr><td colspan="2">[]
[1]
[2]
[1, 2]
[3]
[1, 3]
[2, 3]
[1, 2, 3]</td></tr>
</table>