02 {

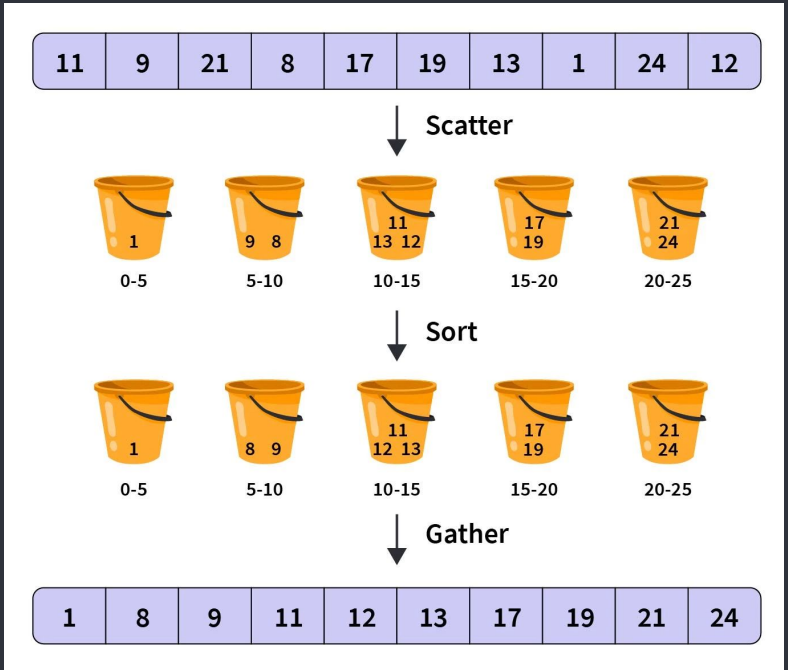
[Bucket Sort]

< Bucket sort is a sorting technique that uses the **Scatter-Gather-Approach** to sort the array. It divides the unsorted array into separate groups and calls them buckets. Sort the individual buckets, and then gather them all together to form the final sorted array.
>

AKA Bin Sort

}

Scatter-Gather Approach



Pseudo code:

```
bucketSorting(arr, n):

    1) Create n empty buckets
    2) loop through each element of the arr and do the following
          Calculate bucketIndex
          Insert the element into the corresponding bucket number

    3) Sort the individual buckets
    4) Gather all the elements together

end bucketSort
```

Working of Bucket Sort

11	9	21	8	17	19	13	1	24	12

Step 1:

Calculating the Range

```
maximumElement = 24
minimumElement = 1
noOfBuckets = 5  // depends upon kind of data
range = (int)(24 - 1) / 5 = 4
```

Working of Bucket Sort

11	9	21	8	17	19	13	1	24	12

Step 2:

Scatter the elements in Buckets

array[0] = 11, Range = [11-15]
array[1] = 9, Range = [6-10]
array[2] = 21, Range = [21-25]
array[3] = 8, Range = [6-10]
And so on...

bucketIndex
= ⌈(array[i] - minimum) / range⌉

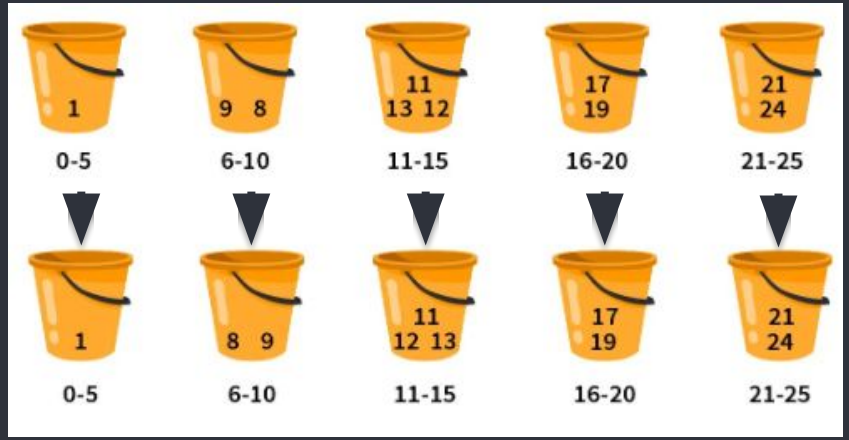
range = 4 calculated in previous slide

Working of Bucket Sort

11	9	21	8	17	19	13	1	24	12

Step 3:

Sort the elements of buckets (using suitable algorithm, depends upon bucket's data structure)



Working of Bucket Sort

11	9	21	8	17	19	13	1	24	12

Step 4:

Gather the data from Buckets

Final Array:

1	8	9	11	12	13	17	19	21	24

Implementation for Floating Number (0.0-1)

```cpp
void bucketSort(float arr[], int n) {    // n: number of buckets
    // Step1: Create n empty buckets
    vector<float> b[n];

    // Step2: Put each element in suitable bucket
    for (int i = 0; i < size; i++) {
        int bi = n * arr[i];  // Index of bucket (Scale up to find the bucket index)
        b[bi].push_back(arr[i]);
    }

    // Step3: Sort all buckets
    for (int i = 0; i < n; i++)
        // by default sort function sort elements in ascending order
        sort(b[i].begin(), b[i].end());

    // Step4: Gather all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];  // updating previous data
}
```

Implementation for Integers:

```
void bucketSort(int arr[], int n, int size) {
    // Step1: Create n empty buckets
    vector<int> b[n];
    int max = *max_element(arr, arr + size);
    int min = *min_element(arr, arr + size);
    int range = (max - min) / n + 1;

    // Step2: Put each element in suitable bucket
    for (int i = 0; i < size; i++) {
        int bi = floor((arr[i] - min) / range);  // index of bucket
        b[bi].push_back(arr[i]);
    }

    // Step3: Sort all buckets
    for (int i = 0; i < n; i++)
        sort(b[i].begin(), b[i].end()); // built-In sort function

    // Step4: Gather all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];  // updating previous data
}
```

Bucket sorting takes linear time, even if the elements are not distributed uniformly.
It holds until the sum of the squares of the bucket sizes is linear in terms of the total number of elements.

Time Complexity		
Best	Equal Distribution in buckets	O(n+k)
Worst	Most of the elements in one bucket	$O(n^2)$
Average		O(n)
Space Complexity	n buckets with k elements	O(n+k)
Stability		Yes