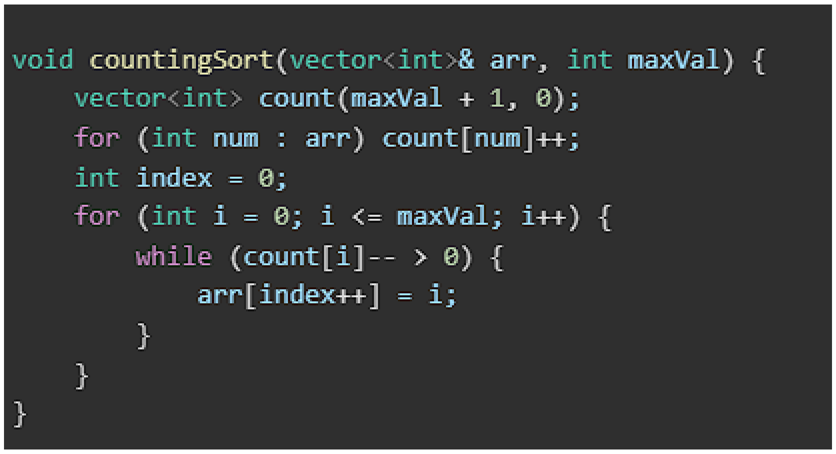
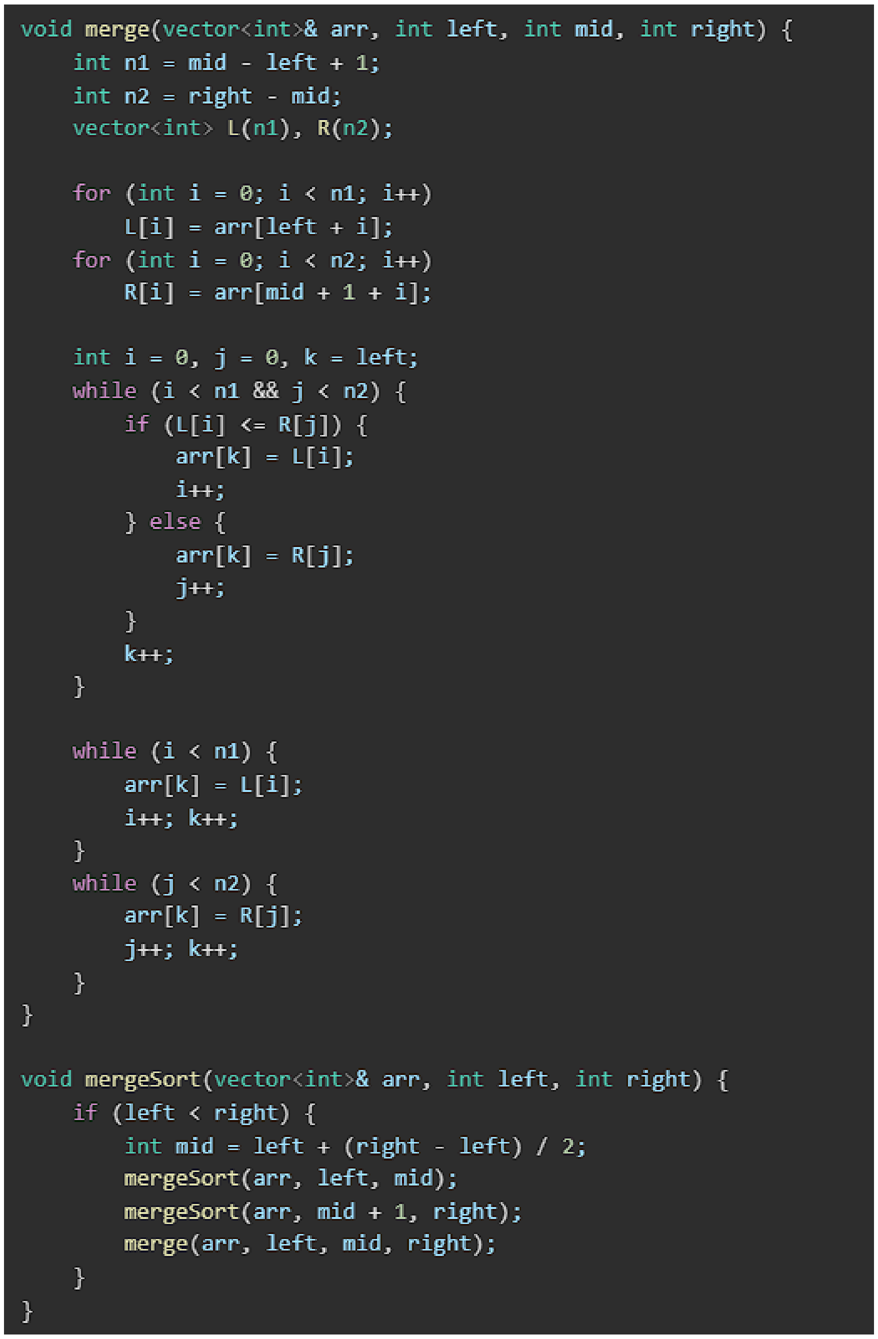
DAA Assignment-2

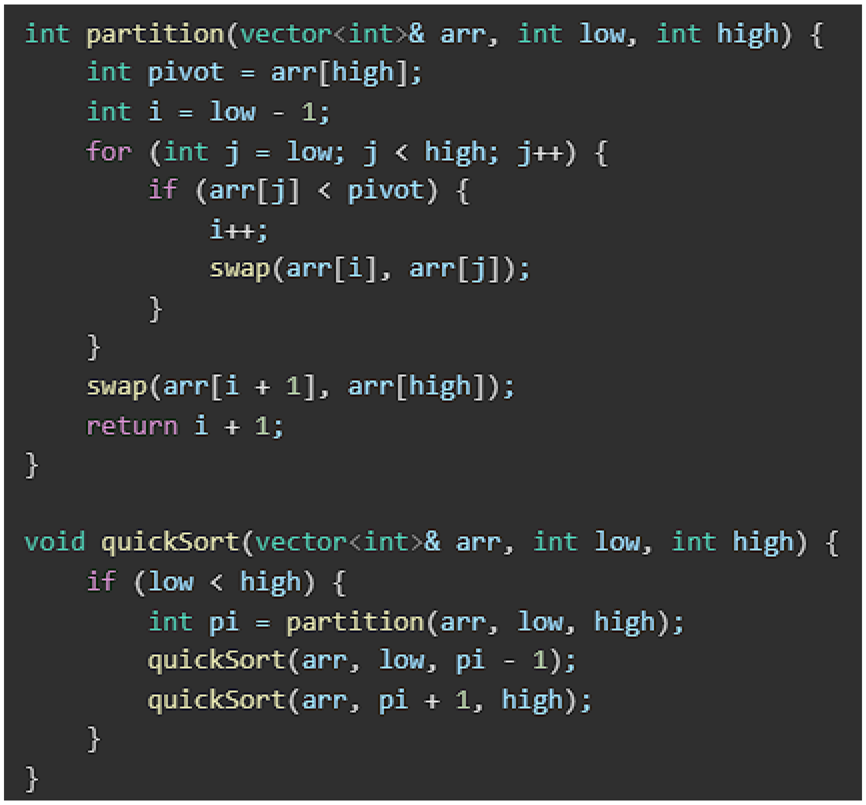
Codes :-



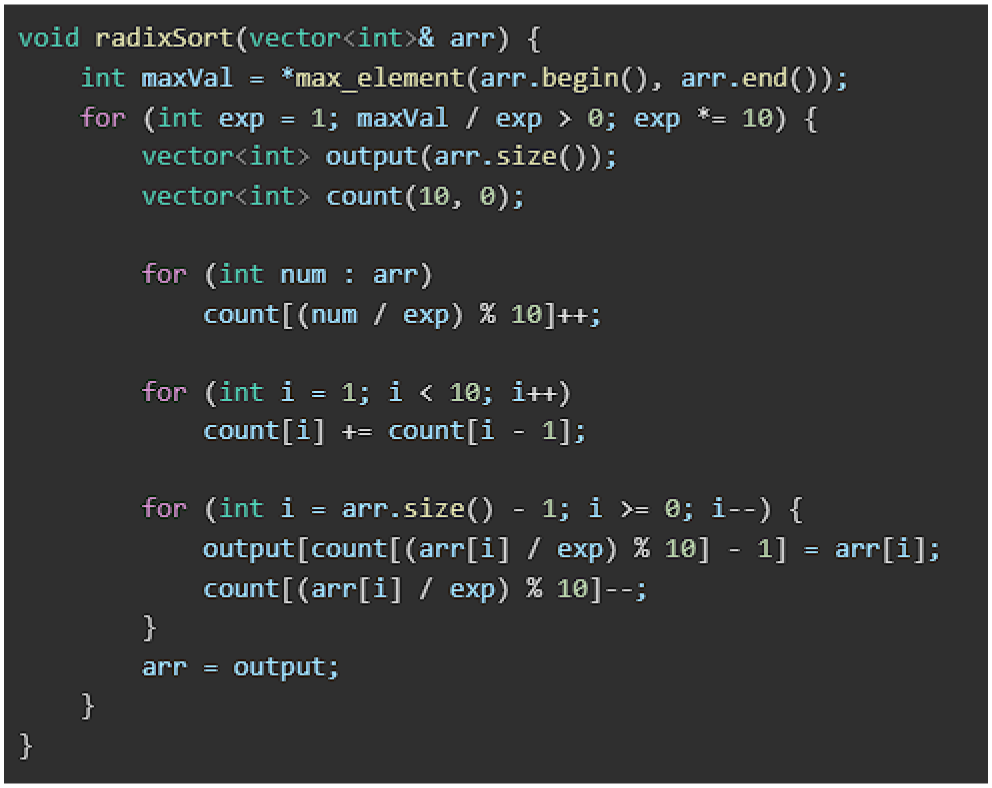
```cpp
void countingSort(vector<int>& arr, int maxVal) {
    vector<int> count(maxVal + 1, 0);
    for (int num : arr) count[num]++;
    int index = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i]-- > 0) {
            arr[index++] = i;
        }
    }
}
```



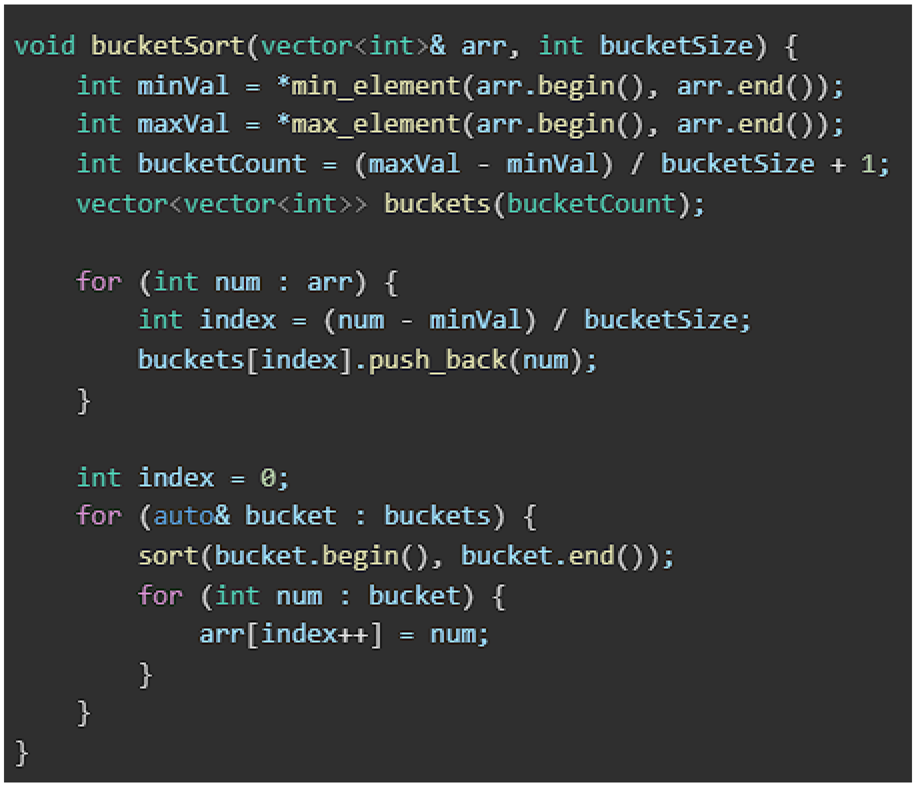
```cpp
void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

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

void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```



```cpp
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

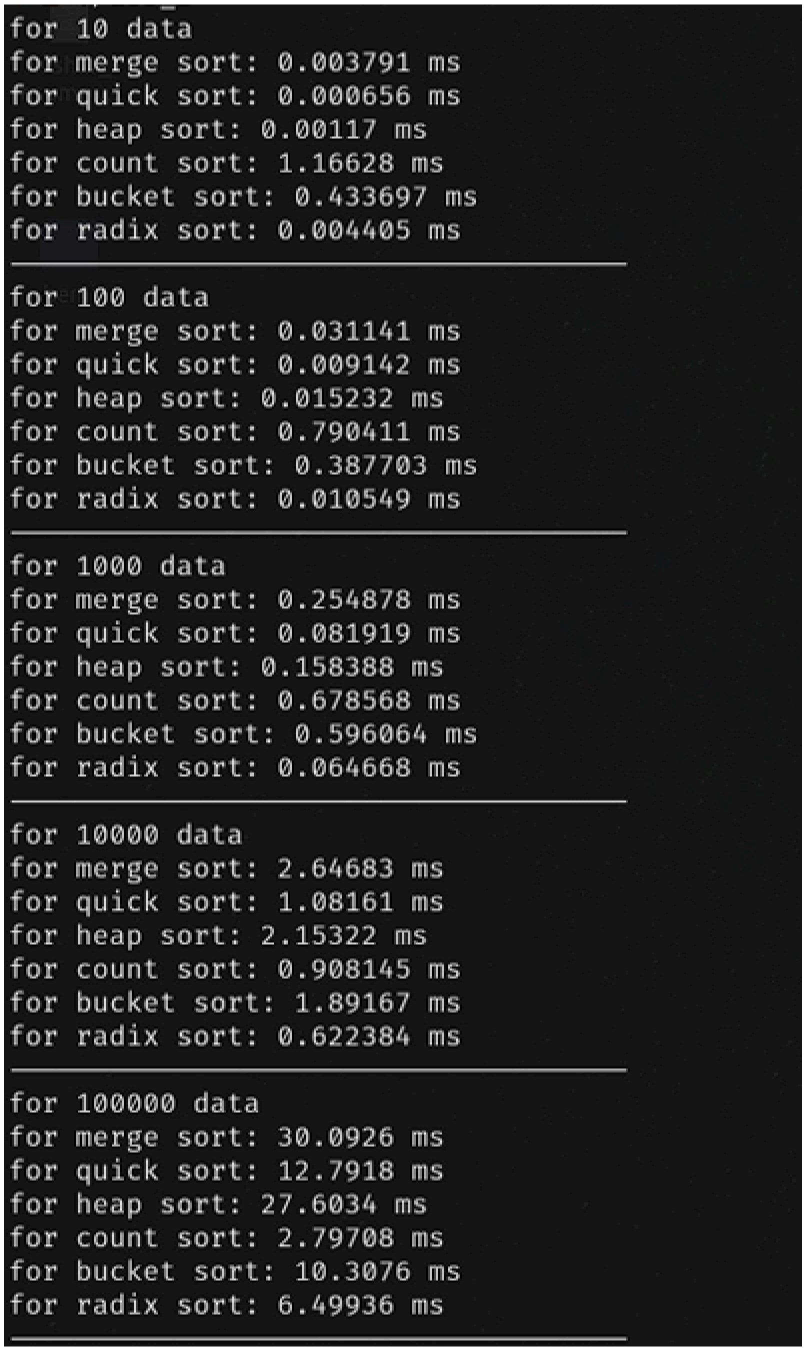


```cpp
void radixSort(vector<int>& arr) {
    int maxVal = *max_element(arr.begin(), arr.end());
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        vector<int> output(arr.size());
        vector<int> count(10, 0);

        for (int num : arr)
            count[(num / exp) % 10]++;

        for (int i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for (int i = arr.size() - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }
        arr = output;
    }
}
```

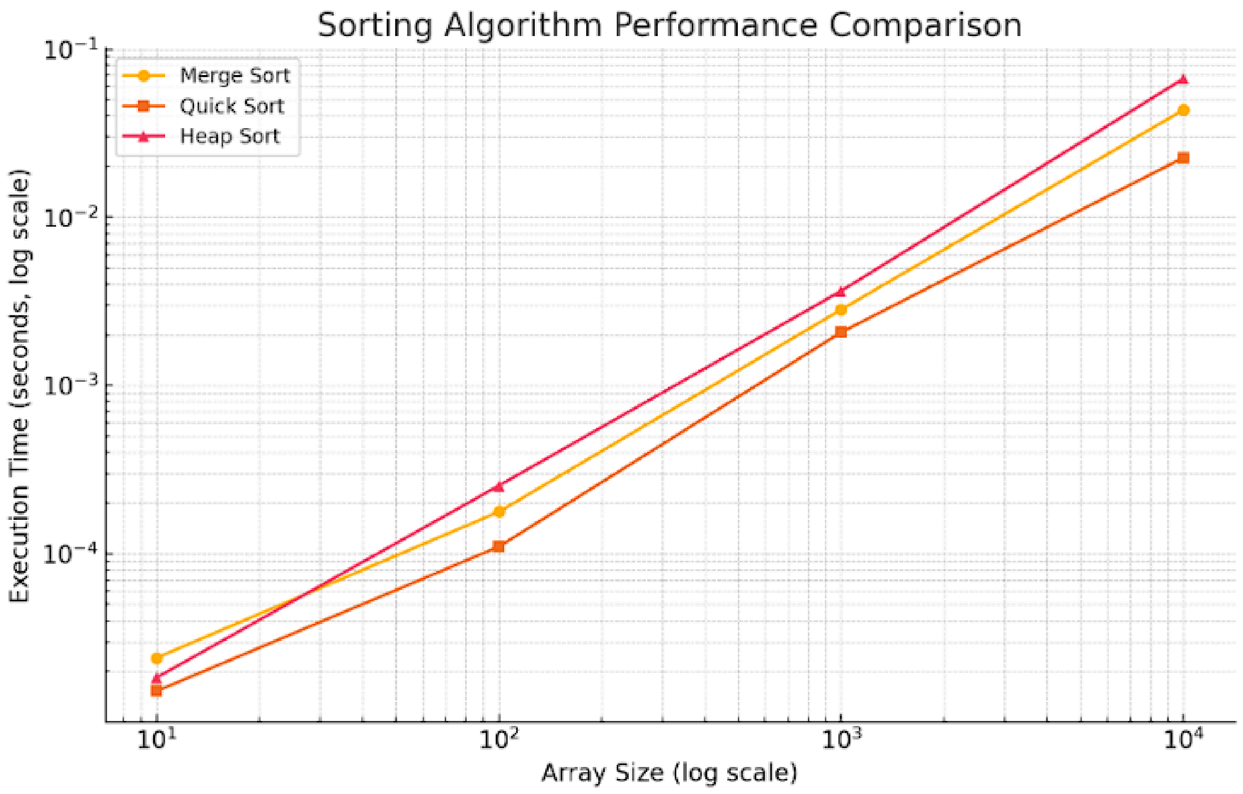


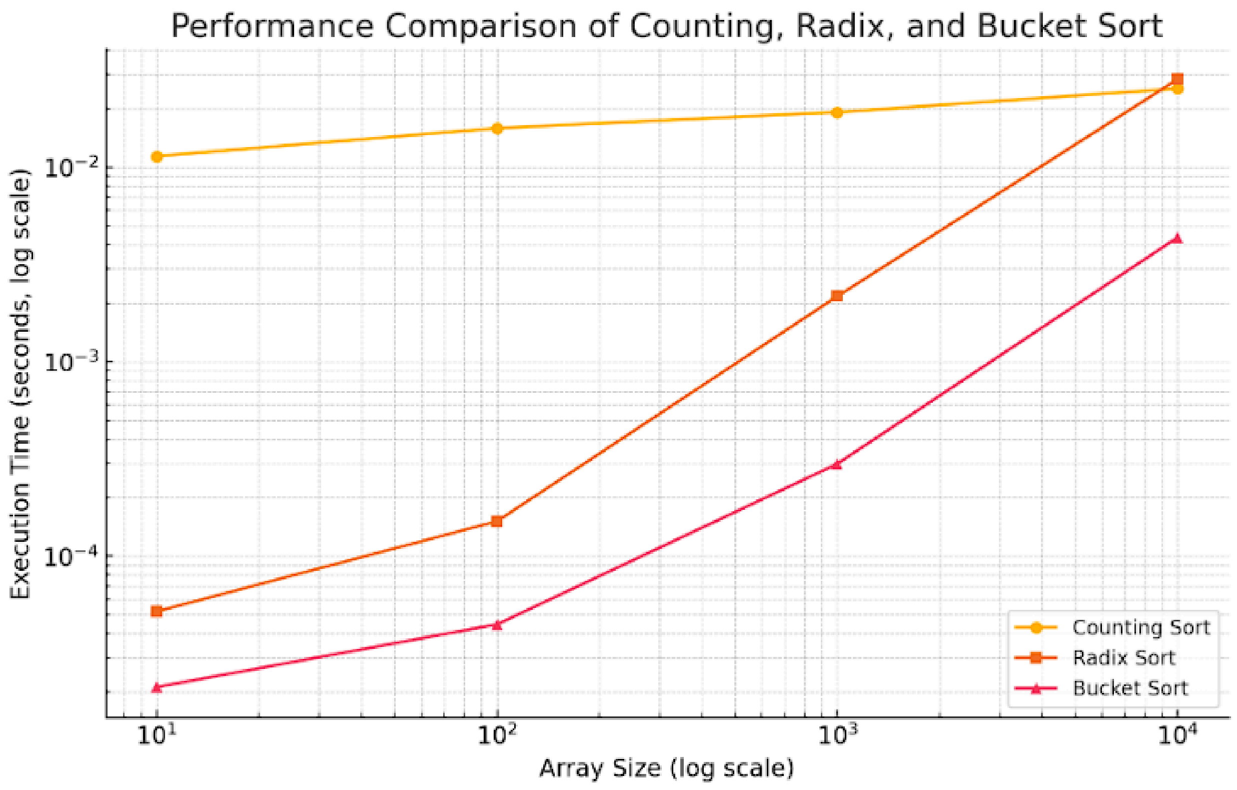
```cpp
void bucketSort(vector<int>& arr, int bucketSize) {
    int minVal = *min_element(arr.begin(), arr.end());
    int maxVal = *max_element(arr.begin(), arr.end());
    int bucketCount = (maxVal - minVal) / bucketSize + 1;
    vector<vector<int>> buckets(bucketCount);

    for (int num : arr) {
        int index = (num - minVal) / bucketSize;
        buckets[index].push_back(num);
    }

    int index = 0;
    for (auto& bucket : buckets) {
        sort(bucket.begin(), bucket.end());
        for (int num : bucket) {
            arr[index++] = num;
        }
    }
}
```

```cpp
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```



```
for 10 data
for merge sort: 0.003791 ms
for quick sort: 0.000656 ms
for heap sort: 0.00117 ms
for count sort: 1.16628 ms
for bucket sort: 0.433697 ms
for radix sort: 0.004405 ms
_____________________________________
for 100 data
for merge sort: 0.031141 ms
for quick sort: 0.009142 ms
for heap sort: 0.015232 ms
for count sort: 0.790411 ms
for bucket sort: 0.387703 ms
for radix sort: 0.010549 ms
_____________________________________
for 1000 data
for merge sort: 0.254878 ms
for quick sort: 0.081919 ms
for heap sort: 0.158388 ms
for count sort: 0.678568 ms
for bucket sort: 0.596064 ms
for radix sort: 0.064668 ms
_____________________________________
for 10000 data
for merge sort: 2.64683 ms
for quick sort: 1.08161 ms
for heap sort: 2.15322 ms
for count sort: 0.908145 ms
for bucket sort: 1.89167 ms
for radix sort: 0.622384 ms
_____________________________________
for 100000 data
for merge sort: 30.0926 ms
for quick sort: 12.7918 ms
for heap sort: 27.6034 ms
for count sort: 2.79708 ms
for bucket sort: 10.3076 ms
for radix sort: 6.49936 ms
_____________________________________
```


Sorting Algorithm Performance Comparison
Merge Sort
Quick Sort
Heap Sort
Execution Time (seconds, log scale)
Array Size (log scale)
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{1}
10^{2}
10^{3}
10^{4}


Performance Comparison of Counting, Radix, and Bucket Sort
Counting Sort
Radix Sort
Bucket Sort
Execution Time (seconds, log scale)
Array Size (log scale)
10^{-2}
10^{-3}
10^{-4}
10^{1}
10^{2}
10^{3}
10^{4}