Question 1: Randomized Quicksort and Duplicate Handling

1.a. Randomized Quicksort Performance

We analyze the performance of a Randomized Quicksort implementation on a sorted array of 100 elements. The algorithm selects a random pivot to avoid the $O(n^2)$ worst-case scenario typical of deterministic quicksort on sorted data.

Table 1: Running times of randomized quicksort over 5 runs (Input Size: 100)

Run	Time (seconds)
Run 1	1.4125×10^{-5}
Run 2	1.0500×10^{-5}
Run 3	1.0959×10^{-5}
Run 4	1.0958×10^{-5}
Run 5	1.0916×10^{-5}
Average	$\mathbf{1.1492 \times 10^{-5}}$

```cpp
// Core logic for Randomized Partitioning
int partition(vector<int> &arr, int low, int high) {
    // Select random pivot and swap to end
    int rand_idx = low + rand() % (high - low + 1);
    swap(arr[rand_idx], arr[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;
}
// (Full wrapper code omitted for brevity, see submission file)
```

Listing 1: Randomized Quicksort Implementation

1.b. Handling Repeated Elements (3-Way Partitioning)

We compare standard Quicksort with a 3-Way Partitioning Quicksort (Dutch National Flag algorithm) on inputs with heavy duplication ($N = 100,000$).

Standard Quicksort Performance

3-Way Partitioning Performance

Analysis: The standard partition scheme degrades significantly ($O(N^2)$) when many elements equal the pivot, as it forces one partition to be empty or nearly empty depending on implementation

Table 2: Standard Quicksort on Repeated Inputs

Input Distribution	Time (seconds)
All 1s (100% duplicates)	18.7253
50% 1s, 50% Random	4.6542
Alternating 2 and 20	9.1439

Table 3: 3-Way Partitioning Quicksort on Repeated Inputs

Input Distribution	Time (seconds)
All 1s (100% duplicates)	**0.0008**
50% 1s, 50% Random	**0.0131**
Alternating 2 and 20	**0.0011**

nuances. 3-Way Partitioning divides the array into three sections: $< P$, $= P$, and $> P$. By isolating duplicate pivots into the middle section and not recursing on them, the effective problem size for recursive calls shrinks drastically. For the "All 1s" case, 3-way partition sorts in $O(N)$ time, as evidenced by the $0.0008s$ runtime vs $18.7s$.

```
void partition(vector<int> &arr, int low, int high, int &l_pivot, int &g_pivot) {
    int pivot = arr[high];
    l_pivot = low;
    g_pivot = high;
    int i = low;

    while (i <= g_pivot) {
        if (arr[i] < pivot) {
            swap(arr[l_pivot++], arr[i++]);
        } else if (arr[i] > pivot) {
            swap(arr[i], arr[g_pivot--]);
        } else {
            i++;
        }
    }
}
```

Listing 2: 3-Way Partitioning Logic

Question 2: Heapsort Implementation

We implement Heapsort using the 'heapify' procedure (percolate down).

Input (Random Permutation): 48 21 55 74 7 13 18 33 19 84 ... (100 elements)

Output (Sorted): 1 2 3 4 5 ... 100

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

    if (left < n && arr[left] > arr[parent]) parent = left;
    if (right < n && arr[right] > arr[parent]) parent = right;
```

```
8
9       if (parent != i) {
10          swap(arr[i], arr[parent]);
11          heapify(arr, n, parent);
12      }
13  }
14
15  void heap_sort(vector<int>& arr) {
16      int n = arr.size();
17      // Build max heap
18      for (int i = n / 2 - 1; i >= 0; i--)
19          heapify(arr, n, i);
20      // Extract elements
21      for (int i = n - 1; i > 0; i--) {
22          swap(arr[0], arr[i]);
23          heapify(arr, i, 0);
24      }
25  }
```

Listing 3: Heapsort Implementation

Question 3: Counting Sort

Counting Sort is a non-comparative integer sorting algorithm with complexity $O(n + k)$.

Input: 20, 18, 5, 7, 16, 10, 9, 3, 12, 14, 0
Sorted: 0, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20

```
1   void counting_sort(vector<int>& arr) {
2       if (arr.empty()) return;
3       auto [min_it, max_it] = minmax_element(arr.begin(), arr.end());
4       int min_val = *min_it;
5       int range = *max_it - min_val + 1;
6
7       vector<int> count(range, 0);
8       vector<int> output(arr.size());
9
10      // Frequency count
11      for (int num : arr) count[num - min_val]++;
12
13      // Cumulative sum
14      for (size_t i = 1; i < count.size(); ++i) count[i] += count[i - 1];
15
16      // Build output (stable sort)
17      for (int i = arr.size() - 1; i >= 0; i--) {
18          output[--count[arr[i] - min_val]] = arr[i];
19      }
20      arr = output;
21  }
```

Listing 4: Counting Sort Implementation

Question 4: Radix Sort

Radix sort processes integers digit by digit (LSD approach here) using Counting Sort as a stable subroutine.

Input: 329, 457, 657, 839, 436, 720, 353
Sorted: 329, 353, 436, 457, 657, 720, 839

```cpp
void digit_counting_sort(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);
    vector<int> count(10, 0);

    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

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

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

void radix_sort(vector<int>& arr) {
    int max_val = *max_element(arr.begin(), arr.end());
    for (int exp = 1; max_val / exp > 0; exp *= 10)
        digit_counting_sort(arr, exp);
}
```

Listing 5: Radix Sort Implementation