

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	1.1492×10^{-5}

```

1 // Core logic for Randomized Partitioning
2 int partition(vector<int> &arr, int low, int high) {
3     // Select random pivot and swap to end
4     int rand_idx = low + rand() % (high - low + 1);
5     swap(arr[rand_idx], arr[high]);
6
7     int pivot = arr[high];
8     int i = low - 1;
9
10    for (int j = low; j < high; j++) {
11        if (arr[j] <= pivot) {
12            i++;
13            swap(arr[i], arr[j]);
14        }
15    }
16    swap(arr[i + 1], arr[high]);
17    return i + 1;
18}
19// (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 nuances. 3-Way Partitioning divides the array into three sections: $< P$, $= P$, and $> P$. By isolating

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

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.

```

1 void partition(vector<int> &arr, int low, int high, int &l_pivot, int &g_pivot) {
2     int pivot = arr[high];
3     l_pivot = low;
4     g_pivot = high;
5     int i = low;
6
7     while (i <= g_pivot) {
8         if (arr[i] < pivot) {
9             swap(arr[l_pivot++], arr[i++]);
10        } else if (arr[i] > pivot) {
11            swap(arr[i], arr[g_pivot--]);
12        } else {
13            i++;
14        }
15    }
16 }
```

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

```

1 void heapify(vector<int>& arr, int n, int i) {
2     int parent = i;
3     int left = 2 * i + 1;
4     int right = 2 * i + 2;
5
6     if (left < n && arr[left] > arr[parent]) parent = left;
7     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

```
1 void digit_counting_sort(vector<int>& arr, int exp) {
2     int n = arr.size();
3     vector<int> output(n);
4     vector<int> count(10, 0);
5
6     for (int i = 0; i < n; i++)
7         count[(arr[i] / exp) % 10]++;
8
9     for (int i = 1; i < 10; i++)
10        count[i] += count[i - 1];
11
12    for (int i = n - 1; i >= 0; i--) {
13        output[--count[(arr[i] / exp) % 10]] = arr[i];
14    }
15    arr = output;
16}
17
18 void radix_sort(vector<int>& arr) {
19     int max_val = *max_element(arr.begin(), arr.end());
20     for (int exp = 1; max_val / exp > 0; exp *= 10)
21         digit_counting_sort(arr, exp);
22}
```

Listing 5: Radix Sort Implementation