Question 1: Selection Algorithms

We implement and compare two algorithms for finding the k-th smallest element in an array of size $N = 100$: Randomized Selection (Rand-Select) and Deterministic Selection (Median of Medians).

1.1. Randomized Selection (Rand-Select)

This algorithm uses a randomized partition scheme similar to Quicksort. It has an expected runtime of $O(N)$ but a worst-case of $O(N^2)$.

```cpp
int randomized_partition(vector<int>& arr, int left, int right, mt19937& g) {
    uniform_int_distribution<int> dist(left, right);
    int pivot_idx = dist(g);

    int pivot = arr[pivot_idx];
    swap(arr[pivot_idx], arr[right]);
    int partition_idx = left;

    for (int i = left; i < right; i++) {
        if (arr[i] < pivot) {
            swap(arr[i], arr[partition_idx++]);
        }
    }
    swap(arr[partition_idx], arr[right]);
    return partition_idx;
}

int rand_select(vector<int>& arr, int p, int q, int i, mt19937& g) {
    if (p == q) return arr[p];
    int r = randomized_partition(arr, p, q, g);
    int k = r - p + 1;
    if (i == k) return arr[r];
    else if (i < k) return rand_select(arr, p, r - 1, i, g);
    else return rand_select(arr, r + 1, q, i - k, g);
}
```

Listing 1: Rand-Select Implementation

Results: Given the input array A (shuffled permutation of $1 \ldots 100$), the algorithm correctly identifies the k-th smallest elements.

Table 1: Results for Rand-Select

k-th Order Statistic	Value
1	1
2	2
3	3
…	…
10	10

1.2. Deterministic Selection (Median of Medians)

This algorithm guarantees $O(N)$ worst-case time by selecting a good pivot (the median of medians of groups of 5).

```cpp
int median_of_medians(vector<int>& arr, int p, int q) {
    int n = q - p + 1;
    if (n <= 5) return median(arr, p, q);

    int groups = (n + 4) / 5;
    vector<int> medians(groups);
    for (int i = 0; i < groups; i++) {
        int l = p + i * 5;
        int r = min(l + 4, q);
        medians[i] = median(arr, l, r);
    }
    return median_of_medians(medians, 0, groups - 1);
}

int select(vector<int>& arr, int p, int q, int i) {
    if (p == q) return arr[p];
    int x = median_of_medians(arr, p, q);
    int pivot_idx = partition(arr, p, q, x);
    int k = pivot_idx - p + 1;

    if (i == k) return arr[pivot_idx];
    else if (i < k) return select(arr, p, pivot_idx - 1, i);
    else return select(arr, pivot_idx + 1, q, i - k);
}
```

Listing 2: Deterministic Select Implementation

Results: The deterministic algorithm also correctly identifies the k-th smallest elements from the shuffled input.

Table 2: Results for Deterministic Select

k-th Order Statistic	Value
1	1
2	2
...	...
10	10

Question 2: Longest Common Subsequence (LCS)

We implement the dynamic programming approach to find **all** Longest Common Subsequences between two strings.

- **Input:** $S1 =$ "ABCBDAB", $S2 =$ "BDCABA"
- **Algorithm:** We construct the DP table $C[i][j]$ storing the length of LCS of $S1[1..i]$ and $S2[1..j]$. To find all subsequences, we backtrack from $C[n][m]$, branching whenever multiple optimal paths ($C[i-1][j] == C[i][j-1]$) exist.

```cpp
void print_all_lcs(const vector<vector<int>>& C, const vector<vector<char>>& b,
                   const string& s1, int i, int j, string curr_lcs,
                   unordered_set<string>& all_lcs) {
    if (i == 0 || j == 0) {
        reverse(curr_lcs.begin(), curr_lcs.end());
        all_lcs.insert(curr_lcs);
        return;
    }
    // If characters match, they must be part of LCS
    if (s1[i-1] == s2[j-1]) { // Logic adapted from b[i][j] == 'D'
        curr_lcs.push_back(s1[i - 1]);
        print_all_lcs(C, b, s1, i - 1, j - 1, curr_lcs, all_lcs);
    } else {
        // Branching logic for multiple paths
        if (C[i - 1][j] == C[i][j])
            print_all_lcs(C, b, s1, i - 1, j, curr_lcs, all_lcs);
        if (C[i][j - 1] == C[i][j])
            print_all_lcs(C, b, s1, i, j - 1, curr_lcs, all_lcs);
    }
}
```

Listing 3: Finding All LCS

Results

```
Length of LCS: 4
Unique LCS found:
1. BCBA
2. BDAB
3. BCAB
```