Section A: Sorting, Linked Lists, and 2D Arrays

Q1: Wallet Sorting
The wallets are already sorted in ascending order. If sorting were necessary, Insertion Sort would b
This is because it's efficient for data that is already sorted or nearly sorted, requiring fewer com

C++
```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;
// Function to perform Insertion Sort
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
int main() {
    vector<int> wallets(20);
    for (int i = 0; i < wallets.size(); ++i) {
        wallets[i] = 20 + i * 10;
    }
    cout << "Original wallet values: ";
    for (int val : wallets) {
        cout << val << " ";
    }
    cout << endl;
    insertionSort(wallets);
    cout << "Wallet values after sorting: ";
    for (int val : wallets) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

Q2: Sorting Wallets with a Limited Range
The best sorting technique for this scenario is Counting Sort. The maximum amount in any wallet is $
the data has a small and limited range. Counting Sort is a non-comparison-based algorithm that's hig
for sorting integers within a small range.

C++
```
#include <iostream>
#include <vector>
#include <map>
using namespace std;
void countingSort(vector<int>& arr) {
    int max_val = 6;
    vector<int> count(max_val + 1, 0);
    for (int num : arr) {
        count[num]++;
    }
    int index = 0;
    for (int i = 0; i <= max_val; ++i) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}
int main() {
    vector<int> wallets = {2, 2, 2, 3, 3, 0, 0, 1, 4};
    cout << "Original wallet values: ";
    for (int val : wallets) {
        cout << val << " ";
    }
    cout << endl;
    countingSort(wallets);
    cout << "Wallet values after sorting: ";
    for (int val : wallets) {
```

```
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

Q3: Student Score Sorting
Quick Sort is the most efficient technique to sort the scores. The problem specifies that the scores
and spread across a wide range, which are ideal conditions for a divide-and-conquer algorithm like Q

C++
```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
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);
    }
}
int main() {
    vector<int> scores = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76};
    cout << "Original scores: ";
    for (int score : scores) {
        cout << score << " ";
    }
    cout << endl;
    quickSort(scores, 0, scores.size() - 1);
    cout << "Scores after sorting: ";
    for (int score : scores) {
        cout << score << " ";
    }
    cout << endl;
    return 0;
}
```

... (Content for Q4 - Q8 and Section B Q1 - Q6 continues here)