Lecture 04: Mergesort and Quicksort

C++ Code Samples — Sedgwick Algorithms Course — lecture-04-samples.cpp

```cpp
// ============================================================================
// Lecture 04: Mergesort and Quicksort
// Sedgwick Algorithms Course
//
// Topics covered:
//   - Top-down mergesort (recursive)
//   - Bottom-up mergesort (iterative)
//   - Quicksort with Lomuto partition scheme
//   - Quicksort with Hoare partition scheme
//   - Median-of-three pivot selection
//   - Demo showing sorted output for each algorithm
// ============================================================================

#include <iostream>
#include <vector>
#include <string>

using namespace std;

// === SECTION: Helper -- Print Array ===
void printArray(const vector<int>& arr, const string& label = "") {
    if (!label.empty()) cout << "  " << label << ": ";
    cout << "[";
    for (int i = 0; i < (int)arr.size(); ++i) {
        if (i > 0) cout << ", ";
        cout << arr[i];
    }
    cout << "]\n";
}

// === SECTION: Top-Down Mergesort (Recursive) ===
// Divide the array in half, recursively sort each half, then merge.
// Guaranteed O(n log n) in all cases. Requires O(n) auxiliary space.

// Merge two sorted halves arr[lo..mid] and arr[mid+1..hi] into arr[lo..hi].
void merge(vector<int>& arr, vector<int>& aux, int lo, int mid, int hi) {
    // Copy to auxiliary array
    for (int k = lo; k <= hi; ++k) aux[k] = arr[k];

    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; ++k) {
        if      (i > mid)          arr[k] = aux[j++];  // left exhausted
        else if (j > hi)           arr[k] = aux[i++];  // right exhausted
        else if (aux[j] < aux[i])  arr[k] = aux[j++];  // right is smaller
        else                       arr[k] = aux[i++];  // left is smaller (stable)
    }
}

void topDownSort(vector<int>& arr, vector<int>& aux, int lo, int hi) {
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    topDownSort(arr, aux, lo, mid);     // sort left half
    topDownSort(arr, aux, mid + 1, hi); // sort right half
    merge(arr, aux, lo, mid, hi);       // merge results
}

void mergesortTopDown(vector<int>& arr) {
    int n = (int)arr.size();
    vector<int> aux(n);
    topDownSort(arr, aux, 0, n - 1);
}
```

```cpp
// === SECTION: Bottom-Up Mergesort (Iterative) ===
// Merge subarrays of size 1, then 2, then 4, ... without recursion.
// Same O(n log n) performance, avoids recursion overhead.

void mergesortBottomUp(vector<int>& arr) {
    int n = arr.size();
    vector<int> aux(n);

    // sz is the size of each subarray being merged
    for (int sz = 1; sz < n; sz *= 2) {
        // lo is the start of the first subarray in each pair
        for (int lo = 0; lo < n - sz; lo += 2 * sz) {
            int mid = lo + sz - 1;
            int hi = min(lo + 2 * sz - 1, n - 1);
            merge(arr, aux, lo, mid, hi);
        }
    }
}


// === SECTION: Quicksort with Lomuto Partition ===
// Lomuto: pivot is the last element. Partition into [<=pivot | pivot | >pivot].
// Simple to understand but does more swaps than Hoare on average.

int lomutoPartition(vector<int>& arr, int lo, int hi) {
    int pivot = arr[hi];  // pivot is the last element
    int i = lo - 1;       // i tracks the boundary of elements <= pivot

    for (int j = lo; j < hi; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[hi]);  // place pivot in its final position
    return i + 1;
}

void quicksortLomuto(vector<int>& arr, int lo, int hi) {
    if (lo >= hi) return;
    int p = lomutoPartition(arr, lo, hi);
    quicksortLomuto(arr, lo, p - 1);
    quicksortLomuto(arr, p + 1, hi);
}

// === SECTION: Quicksort with Hoare Partition ===
// Hoare: pivot is the first element. Two pointers scan inward.
// Fewer swaps on average than Lomuto. Original quicksort partition.

int hoarePartition(vector<int>& arr, int lo, int hi) {
    int pivot = arr[lo];
    int i = lo - 1;
    int j = hi + 1;

    while (true) {
        // Move i right, skipping elements less than pivot
        do ++i; while (arr[i] < pivot);
        // Move j left, skipping elements greater than pivot
        do --j; while (arr[j] > pivot);

        if (i >= j) return j;  // pointers crossed: partition done
        swap(arr[i], arr[j]);
    }
}
```

```cpp
void quicksortHoare(vector<int>& arr, int lo, int hi) {
    if (lo >= hi) return;
    int p = hoarePartition(arr, lo, hi);
    quicksortHoare(arr, lo, p);        // note: includes p (Hoare property)
    quicksortHoare(arr, p + 1, hi);
}

// === SECTION: Median-of-Three Pivot Selection ===
// Chooses the median of first, middle, and last elements as pivot.
// Avoids worst-case O(n^2) on sorted or reverse-sorted input.

int medianOfThree(vector<int>& arr, int lo, int hi) {
    int mid = lo + (hi - lo) / 2;

    // Sort the three elements so arr[lo] <= arr[mid] <= arr[hi]
    if (arr[mid] < arr[lo])  swap(arr[lo], arr[mid]);
    if (arr[hi]  < arr[lo])  swap(arr[lo], arr[hi]);
    if (arr[hi]  < arr[mid]) swap(arr[mid], arr[hi]);

    // Place median (arr[mid]) at position hi-1 as the pivot
    swap(arr[mid], arr[hi - 1]);
    return arr[hi - 1];
}

int medianOfThreePartition(vector<int>& arr, int lo, int hi) {
    if (hi - lo < 3) {
        // Too few elements for median-of-three; use simple Lomuto
        return lomutoPartition(arr, lo, hi);
    }
    int pivot = medianOfThree(arr, lo, hi);
    int i = lo;
    int j = hi - 1;   // pivot is at hi-1 after medianOfThree

    while (true) {
        while (arr[++i] < pivot) {}
        while (arr[--j] > pivot) {}
        if (i >= j) break;
        swap(arr[i], arr[j]);
    }
    swap(arr[i], arr[hi - 1]);   // restore pivot
    return i;
}

void quicksortMedian3(vector<int>& arr, int lo, int hi) {
    if (lo >= hi) return;
    int p = medianOfThreePartition(arr, lo, hi);
    quicksortMedian3(arr, lo, p - 1);
    quicksortMedian3(arr, p + 1, hi);
}

// === SECTION: Partition Trace ===
// Shows one level of Lomuto partitioning for educational purposes.
void partitionTrace(vector<int> arr) {
    cout << "\n--- Lomuto Partition Trace ---\n";
    printArray(arr, "Input");
    int lo = 0, hi = (int)arr.size() - 1;
    cout << "  Pivot (last element): " << arr[hi] << "\n";

    int pivotIdx = lomutoPartition(arr, lo, hi);

    cout << "  After partition (pivot at index " << pivotIdx << "):\n";
    cout << "    Left  (<=pivot): [";
    for (int i = lo; i < pivotIdx; i++) {
```

```
        if (i > lo) cout << ", ";
        cout << arr[i];
    }
    cout << "]\n";
    cout << "    Pivot:           " << arr[pivotIdx] << "\n";
    cout << "    Right (>pivot):  [";
    for (int i = pivotIdx + 1; i <= hi; ++i) {
        if (i > pivotIdx + 1) cout << ", ";
        cout << arr[i];
    }
    cout << "]\n";
}

// === MAIN ===
int main() {
    cout << "==========================================\n";
    cout << " Lecture 04: Mergesort and Quicksort\n";
    cout << "==========================================\n";

    vector<int> original = {38, 27, 43, 3, 9, 82, 10};

    // --- Top-Down Mergesort ---
    cout << "\n--- Top-Down Mergesort (Recursive) ---\n";
    vector<int> a = original;
    printArray(a, "Input");
    mergesortTopDown(a);
    printArray(a, "Sorted");

    // --- Bottom-Up Mergesort ---
    cout << "\n--- Bottom-Up Mergesort (Iterative) ---\n";
    vector<int> b = original;
    printArray(b, "Input");
    mergesortBottomUp(b);
    printArray(b, "Sorted");

    // --- Quicksort (Lomuto) ---
    cout << "\n--- Quicksort (Lomuto Partition) ---\n";
    vector<int> c = original;
    printArray(c, "Input");
    quicksortLomuto(c, 0, (int)c.size() - 1);
    printArray(c, "Sorted");

    // --- Quicksort (Hoare) ---
    cout << "\n--- Quicksort (Hoare Partition) ---\n";
    vector<int> d = original;
    printArray(d, "Input");
    quicksortHoare(d, 0, (int)d.size() - 1);
    printArray(d, "Sorted");

    // --- Quicksort (Median-of-Three) ---
    cout << "\n--- Quicksort (Median-of-Three Pivot) ---\n";
    vector<int> e = original;
    printArray(e, "Input");
    quicksortMedian(e, 0, (int)e.size() - 1);
    printArray(e, "Sorted");

    // --- Partition Trace ---
    partitionTrace({15, 3, 9, 8, 5, 2, 7, 1, 6});

    // --- All produce the same result ---
    cout << "\n--- Verification: All Algorithms Agree ---\n";
    bool allMatch = (a == b && b == c && c == d && d == e);
    cout << "  All 5 sorted arrays identical: "
         << (allMatch ? "YES" : "NO") << "\n";
```

```
    // --- Larger demo with sorted input (worst case for naive quicksort) ---
    cout << "\n--- Sorted Input Demo (n=20) ---\n";
    vector<int> sorted20(20);
    for (int i = 0; i < 20; ++i) sorted20[i] = i + 1;
    printArray(sorted20, "Input (already sorted)");

    vector<int> s1 = sorted20, s2 = sorted20, s3 = sorted20;
    mergesortTopDown(s1);
    quicksortHoare(s2, 0, 19);
    quicksortMedian3(s3, 0, 19);

    printArray(s1, "Mergesort result");
    printArray(s2, "Quicksort (Hoare) result");
    printArray(s3, "Quicksort (Median-of-3) result");
    cout << "  Note: Lomuto quicksort degrades to O(n^2) on sorted input.\n";
    cout << "  Median-of-three and Hoare handle sorted input much better.\n";

    return 0;
}
```