MTH 4300: Algorithms, Computers and Programming II

Fall 2025

Section: STRA

Problem Set 3 - Solutions

Due Date: September 23, 2025

Iterators and Algorithms (20 points)

Part 1: Basic Day Statistics

```cpp
#include <vector>
#include <algorithm>
#include <numeric>
#include <cmath>

std::vector<double> calculateDayStats(const std::vector<double>& minute_prices) {
    if (minute_prices.empty()) {
        return {0.0, 0.0, 0.0, 0.0};
    }

    // Find min and max using STL algorithms
    auto min_it = std::min_element(minute_prices.begin(), minute_prices.end());
    auto max_it = std::max_element(minute_prices.begin(), minute_prices.end());
    double min_price = *min_it;
    double max_price = *max_it;

    // Calculate average using std::accumulate
    double sum = std::accumulate(minute_prices.begin(), minute_prices.end(), 0.0);
    double average = sum / minute_prices.size();

    // Calculate standard deviation
    std::vector<double> squared_diffs(minute_prices.size());
    std::transform(minute_prices.begin(), minute_prices.end(),
                   squared_diffs.begin(),
                   [average](double price) {
                       double diff = price - average;
                       return diff * diff;
                   });

    double variance = std::accumulate(squared_diffs.begin(), squared_diffs.end(), 0.0)
                      / minute_prices.size();
    double std_deviation = std::sqrt(variance);

    return {min_price, max_price, average, std_deviation};
}
```

Part 2: K-Session Analysis

```cpp
std::vector<double> analyzeKSessions(const std::vector<double>& minute_prices, int k) {
    if (minute_prices.empty() || k <= 0) {
        return {};
    }

    std::vector<double> result;
    int session_size = minute_prices.size() / k;
    double highest_stddev = -1.0;
    int highest_session = 1;

    for (int i = 0; i < k; ++i) {
        auto start_it = minute_prices.begin() + (i * session_size);
        auto end_it = (i == k - 1) ? minute_prices.end() :
                      minute_prices.begin() + ((i + 1) * session_size);

        // Create session vector using iterators
        std::vector<double> session(start_it, end_it);

        if (session.empty()) {
            result.push_back(0.0);
            continue;
        }
```

```
        // Calculate session statistics
        double session_sum = std::accumulate(session.begin(), session.end(), 0.0);
        double session_avg = session_sum / session.size();

        std::vector<double> session_squared_diffs(session.size());
        std::transform(session.begin(), session.end(),
                       session_squared_diffs.begin(),
                       [session_avg](double price) {
                           double diff = price - session_avg;
                           return diff * diff;
                       });

        double session_variance = std::accumulate(session_squared_diffs.begin(),
                                                  session_squared_diffs.end(), 0.0)
                                  / session.size();
        double session_stddev = std::sqrt(session_variance);

        result.push_back(session_stddev);

        if (session_stddev > highest_stddev) {
            highest_stddev = session_stddev;
            highest_session = i + 1;
        }
    }

    result.push_back(static_cast<double>(highest_session));
    return result;
}
```

Recursion (20 points)

Question 1: Single Variable Recursion

```
// Helper function to sum digits
int sumDigits(int n) {
    if (n == 0) {
        return 0;
    }
    return (n % 10) + sumDigits(n / 10);
}

int digitalRoot(int n) {
    // Base case: single digit
    if (n < 10) {
        return n;
    }

    // Recursive case: sum digits and find digital root of sum
    int digit_sum = sumDigits(n);
    return digitalRoot(digit_sum);
}
```

Alternative approach without helper function:

```
int digitalRoot(int n) {
    // Base case: single digit
    if (n < 10) {
        return n;
    }

    // Calculate sum of digits
    int sum = 0;
    int temp = n;
    while (temp > 0) {
        sum += temp % 10;
        temp /= 10;
    }

    // Recursive case
    return digitalRoot(sum);
}
```

Question 2: Multi-Variable Recursion

```
int gcd(int a, int b) {
    // Base case: if b is 0, then a is the GCD
    if (b == 0) {
```

```
        return a;
    }

    // Recursive case: gcd(a, b) = gcd(b, a % b)
    return gcd(b, a % b);
}
```

Test cases for verification:

```
#include <cassert>

void testSolutions() {
    // Test digitalRoot
    assert(digitalRoot(9) == 9);
    assert(digitalRoot(38) == 2);
    assert(digitalRoot(1234) == 1);
    assert(digitalRoot(9999) == 9);

    // Test gcd
    assert(gcd(48, 18) == 6);
    assert(gcd(15, 25) == 5);
    assert(gcd(17, 13) == 1);
    assert(gcd(100, 25) == 25);

    // Test calculateDayStats
    std::vector<double> prices = {100.0, 102.0, 98.0, 105.0, 97.0, 103.0};
    auto stats = calculateDayStats(prices);
    assert(std::abs(stats[0] - 97.0) < 0.01);   // min
    assert(std::abs(stats[1] - 105.0) < 0.01);  // max
    assert(std::abs(stats[2] - 100.833) < 0.01); // avg
    assert(std::abs(stats[3] - 2.99) < 0.1);    // stddev
}
```