MTH 4300: Algorithms, Computers and Programming II

Fall 2025

Section: STRA

Problem Set 3

Due Date: September 23, 2025

Iterators and Algorithms (20 points)

You're analyzing a full trading day of minute-by-minute stock prices to calculate key statistics and identify trading sessions. This question has two parts that build on each other.

Part 1: Basic Day Statistics

Function Signature:

```
std::vector<double> calculateDayStats(const std::vector<double>& minute_prices);
```

Formula for standard deviation:

$\sigma = \sqrt{\left(\frac{1}{N}\right) \cdot \sum_{i=0}^{N} (\text{prices}_i - \mu)^2}$

Requirements:
1. **Use iterators** - No index-based access
2. **Use STL algorithms** - Must use at least 3 different algorithms from `<algorithm>`
3. **Return format** - Vector with exactly 4 values: `[min_price, max_price, average_price, std_deviation]`

Example:

```
// Trading day with 6 minute prices
std::vector<double> prices = {100.0, 102.0, 98.0, 105.0, 97.0, 103.0};
std::vector<double> result = calculateDayStats(prices);
// Expected result: [97.0, 105.0, 100.833, 2.99]
// Min: 97.0, Max: 105.0, Average: 100.833, StdDev: 2.99
```

Part 2: K-Session Analysis

Function Signature:

```
std::vector<double> analyzeKSessions(const std::vector<double>& minute_prices, int k);
```

Requirements: Split the trading day into `k` equal sessions and find which session had the highest volatility (standard deviation).

1. **Use iterators** - No index-based access allowed
2. **Use STL algorithms** - Must use algorithms to process each session
3. **Return format** - Vector with `k+1` values: `[session1_stddev, session2_stddev, ..., sessionK_stddev, highest_session_number]`

Session Rules:
- Divide total minutes into `k` equal parts
- Return session number (1, 2, ..., k) that has highest standard deviation
- If multiple sessions tie for highest volatility, return the first one

Examples:

```
// Example 1: k=3, 9 prices → 3 prices per session
std::vector<double> prices1 = {100.0, 102.0, 98.0, 105.0, 97.0, 103.0, 101.0, 99.0, 104.0};
// Session 1: [100.0, 102.0, 98.0] → std_dev ≈ 1.63
// Session 2: [105.0, 97.0, 103.0] → std_dev ≈ 3.27
// Session 3: [101.0, 99.0, 104.0] → std_dev ≈ 2.05
// Expected: [1.63, 3.27, 2.05, 2.0] (session 2 has highest volatility)
```

Recursion (20 points)

Question 1: Single Variable Recursion

Write a recursive function that calculates the digital root of a positive integer. The digital root is obtained by repeatedly summing the digits until a single digit remains.

Function Signature:

```
int digitalRoot(int n);
```

Requirements:
1. **Use recursion** - No loops allowed
2. **Base case** - Single digit numbers (0-9) return themselves
3. **Recursive case** - Sum all digits, then recursively find digital root of the sum
4. **Helper function** - You may create a helper function to sum digits

Algorithm:
- If n $<$ 10, return n (base case)
- Otherwise, sum all digits of n and recursively call `digitalRoot` on the sum

Example:

```
assert(digitalRoot(9) == 9);        // Single digit
assert(digitalRoot(38) == 2);       // 3+8=11, 1+1=2
assert(digitalRoot(1234) == 1);     // 1+2+3+4=10, 1+0=1
assert(digitalRoot(9999) == 9);     // 9+9+9+9=36, 3+6=9
```

Question 2: Multi-Variable Recursion

Write a recursive function that calculates the number of paths in a grid from top-left (0,0) to bottom-right (m,n) where you can only move right or down.

Function Signature:

```
int countPaths(int m, int n);
```

Requirements:
1. **Use recursion** - No loops or dynamic programming allowed
2. **Base cases** - If `m == 0` or `n == 0`, there's exactly 1 path (straight line)
3. **Recursive case** - Total paths = paths from `(m-1,n)` + paths from `(m,n-1)`
4. **Grid coordinates** - m represents remaining moves right, n represents remaining moves down

Algorithm:
- To reach `(m,n)`, you must come from either `(m-1,n)` or `(m,n-1)`
- Total paths = `countPaths(m-1,n) + countPaths(m,n-1)`
- Base case: `countPaths(0,n) = countPaths(m,0) = 1`

Example:

```
assert(countPaths(1, 1) == 2);      // 2x2 grid: RD or DR
assert(countPaths(2, 2) == 6);      // 3x3 grid: multiple paths
assert(countPaths(3, 2) == 10);     // 4x3 grid
assert(countPaths(0, 5) == 1);      // Straight line (edge case)
```