MTH 4300: Algorithms, Computers and Programming II

Fall 2025

Section: STRA

Midterm 1 Practice Exam Solutions (Oct 2, 2025)

Question 1 Solution

Tracing through the execution:

Initial state:
- `data = {3, 8, 5, 12, 7}`
- `factor = 2`

First call to `modifyArray(data, &factor)`:
- i=0: `arr[0] = 3` (odd), so `3 + 2 = 5`
- i=1: `arr[1] = 8` (even), so `8 * 2 = 16`
- i=2: `arr[2] = 5` (odd), so `5 + 2 = 7`
- i=3: `arr[3] = 12` (even), so `12 * 2 = 24`
- i=4: `arr[4] = 7` (odd), so `7 + 2 = 9`
- `(*multiplier)++` so `factor = 3`

Second call to `modifyArray(data, &factor)`:
- i=0: `arr[0] = 5` (odd), so `5 + 3 = 8`
- i=1: `arr[1] = 16` (even), so `16 * 3 = 48`
- i=2: `arr[2] = 7` (odd), so `7 + 3 = 10`
- i=3: `arr[3] = 24` (even), so `24 * 3 = 72`
- i=4: `arr[4] = 9` (odd), so `9 + 3 = 12`
- `(*multiplier)++` so `factor = 4`

Output:

```
Initial: 3 8 5 12 7 factor=2
After first call: 5 16 7 24 9 factor=3
After second call: 8 48 10 72 12 factor=4
```

Question 2 Solution

Tracing through recursive calls:

Call: `transform(6, 2)`
- `n=6` (even), `depth=2`
- Calls: `transform(3, 1) + transform(3, 1)`

Call: `transform(3, 1)` (first)
- `n=3` (odd), `depth=1`
- Calls: `transform(2, 0) + 1`

Call: `transform(2, 0)`
- `depth=0`, base case reached, returns `2`

Back to `transform(3, 1)` (first):
- Returns `2 + 1 = 3`

Call: `transform(3, 1)` (second)
- Same as first call, returns `3`

Back to `transform(6, 2)`:
- Returns `3 + 3 = 6`

Output:

```
Called transform(6, 2)
Called transform(3, 1)
Called transform(2, 0)
Base case reached, returning 2
```

```
Odd case: returning 3
Called transform(3, 1)
Called transform(2, 0)
Base case reached, returning 2
Odd case: returning 3
Even case: returning 6
Final result: 6
```

Question 3 Solution

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

int findSecondLargest(const vector<int>& arr) {
    if (arr.empty()) {
        return -1;
    }

    int largest = INT_MIN;
    int secondLargest = INT_MIN;

    for (int num : arr) {
        if (num > largest) {
            secondLargest = largest;
            largest = num;
        } else if (num > secondLargest && num < largest) {
            secondLargest = num;
        }
    }

    return (secondLargest == INT_MIN) ? -1 : secondLargest;
}

int main() {
    // Test case 1: Normal case
    vector<int> test1 = {5, 2, 8, 2, 9, 1};
    cout << "Test 1: {5, 2, 8, 2, 9, 1} -> " << findSecondLargest(test1) << endl;

    // Test case 2: All same elements
    vector<int> test2 = {3, 3, 3};
    cout << "Test 2: {3, 3, 3} -> " << findSecondLargest(test2) << endl;

    // Test case 3: Two unique elements
    vector<int> test3 = {7, 7, 5};
    cout << "Test 3: {7, 7, 5} -> " << findSecondLargest(test3) << endl;

    // Test case 4: Single element
    vector<int> test4 = {42};
    cout << "Test 4: {42} -> " << findSecondLargest(test4) << endl;

    // Test case 5: Empty array
    vector<int> test5 = {};
    cout << "Test 5: {} -> " << findSecondLargest(test5) << endl;

    return 0;
}
```

Question 4 Solution

```cpp
#include <iostream>
using namespace std;

void reverseSegment(int* start, int* end) {
    while (start < end) {
        int temp = *start;
        *start = *end;
        *end = temp;
        start++;
        end--;
    }
}

void reverseSegments(int* arr, int size, int segmentSize) {
    if (segmentSize <= 1 || segmentSize >= size) {
        return;
    }

    int* current = arr;
    int remaining = size;

    while (remaining > 0) {
        int currentSegmentSize = (remaining >= segmentSize) ? segmentSize : remaining;
        reverseSegment(current, current + currentSegmentSize - 1);
        current += currentSegmentSize;
        remaining -= currentSegmentSize;
    }
}

void printArray(int* arr, int size) {
    cout << "{";
    for (int i = 0; i < size; i++) {
        cout << *(arr + i);
        if (i < size - 1) cout << ", ";
    }
    cout << "}" << endl;
}

int main() {
    // Test case 1: Normal case
    int arr1[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    int size1 = 9;
    cout << "Test 1 - Before: ";
    printArray(arr1, size1);
    reverseSegments(arr1, size1, 3);
    cout << "Test 1 - After (segment size 3): ";
    printArray(arr1, size1);
    cout << endl;

    // Test case 2: Uneven segments
    int arr2[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int size2 = 8;
    cout << "Test 2 - Before: ";
    printArray(arr2, size2);
    reverseSegments(arr2, size2, 3);
    cout << "Test 2 - After (segment size 3): ";
    printArray(arr2, size2);
    cout << endl;
```

```
    // Test case 3: Edge case - segment size = 1
    int arr3[] = {1, 2, 3, 4, 5};
    int size3 = 5;
    cout << "Test 3 - Before: ";
    printArray(arr3, size3);
    reverseSegments(arr3, size3, 1);
    cout << "Test 3 - After (segment size 1): ";
    printArray(arr3, size3);
    cout << endl;

    return 0;
}
```

Question 5 Solution

```
#include <iostream>
#include <vector>
using namespace std;

bool canPartition(const vector<int>& arr, int index, int sum1, int sum2) {
    // Base case: reached end of array
    if (index == arr.size()) {
        return sum1 == sum2;
    }

    // Try adding current element to subset 1
    bool option1 = canPartition(arr, index + 1, sum1 + arr[index], sum2);

    // Try adding current element to subset 2
    bool option2 = canPartition(arr, index + 1, sum1, sum2 + arr[index]);

    // Return true if either option works
    return option1 || option2;
}

bool canPartition(const vector<int>& arr) {
    if (arr.empty()) {
        return true;
    }
    return canPartition(arr, 0, 0, 0);
}

int main() {
    // Test case 1: Can be partitioned
    vector<int> test1 = {1, 5, 11, 5};
    cout << "Test 1: {1, 5, 11, 5} -> " << (canPartition(test1) ? "true" : "false") <<
endl;

    // Test case 2: Cannot be partitioned
    vector<int> test2 = {1, 2, 3, 5};
    cout << "Test 2: {1, 2, 3, 5} -> " << (canPartition(test2) ? "true" : "false") <<
endl;

    // Test case 3: Simple case
    vector<int> test3 = {2, 2};
    cout << "Test 3: {2, 2} -> " << (canPartition(test3) ? "true" : "false") << endl;

    // Test case 4: Single element
    vector<int> test4 = {5};
    cout << "Test 4: {5} -> " << (canPartition(test4) ? "true" : "false") << endl;
```

```
    return 0;
}
```

Question 6 Solution

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> spiralTraversal(const vector<vector<int>>& matrix) {
    vector<int> result;

    if (matrix.empty() || matrix[0].empty()) {
        return result;
    }

    int rows = matrix.size();
    int cols = matrix[0].size();

    int top = 0, bottom = rows - 1;
    int left = 0, right = cols - 1;

    while (top <= bottom && left <= right) {
        // Traverse right on top row
        for (int j = left; j <= right; j++) {
            result.push_back(matrix[top][j]);
        }
        top++;

        // Traverse down on right column
        for (int i = top; i <= bottom; i++) {
            result.push_back(matrix[i][right]);
        }
        right--;

        // Traverse left on bottom row (if we still have rows)
        if (top <= bottom) {
            for (int j = right; j >= left; j--) {
                result.push_back(matrix[bottom][j]);
            }
            bottom--;
        }

        // Traverse up on left column (if we still have columns)
        if (left <= right) {
            for (int i = bottom; i >= top; i--) {
                result.push_back(matrix[i][left]);
            }
            left++;
        }
    }

    return result;
}

void printMatrix(const vector<vector<int>>& matrix) {
    for (const auto& row : matrix) {
        for (int val : row) {
            cout << val << "\t";
```

```
        }
        cout << endl;
    }
}

void printVector(const vector<int>& vec) {
    cout << "{";
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i];
        if (i < vec.size() - 1) cout << ", ";
    }
    cout << "}" << endl;
}

int main() {
    // Test case 1: 3x4 matrix
    vector<vector<int>> matrix1 = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    cout << "Test 1 - Matrix:" << endl;
    printMatrix(matrix1);
    cout << "Spiral traversal: ";
    printVector(spiralTraversal(matrix1));
    cout << endl;

    // Test case 2: 2x2 matrix
    vector<vector<int>> matrix2 = {
        {1, 2},
        {3, 4}
    };
    cout << "Test 2 - Matrix:" << endl;
    printMatrix(matrix2);
    cout << "Spiral traversal: ";
    printVector(spiralTraversal(matrix2));
    cout << endl;

    // Test case 3: Single row
    vector<vector<int>> matrix3 = {{1, 2, 3, 4, 5}};
    cout << "Test 3 - Matrix:" << endl;
    printMatrix(matrix3);
    cout << "Spiral traversal: ";
    printVector(spiralTraversal(matrix3));
    cout << endl;

    // Test case 4: Single column
    vector<vector<int>> matrix4 = {{1}, {2}, {3}, {4}};
    cout << "Test 4 - Matrix:" << endl;
    printMatrix(matrix4);
    cout << "Spiral traversal: ";
    printVector(spiralTraversal(matrix4));
    cout << endl;

    return 0;
}
```