MTH 4300: Algorithms, Computers and Programming II

Spring 2026

Section: SMWA

Midterm 1 Practice Exam 2 — Solution Key

Instructions:

- **Tracing:** Write the exact output, showing intermediate variable values step by step.
- **Debugging:** For each bug — state the line, explain the problem, and write the corrected code.
- **Programming:** Write pseudocode in Part A first, then a complete C++ implementation in Part B.

Note: Submit handwritten responses as a PDF.

Question 1: Tracing — Structs & Pointers

What is the output of the following program? Trace through step by step, tracking the state of `a`, `b`, and `ptr` after each statement.

```cpp
#include <iostream>

struct Point {
    int x;
    int y;
};

void shift(Point* p, int dx, int dy) {
    p->x += dx;
    p->y += dy;
}

int main() {
    Point a = {3, 4};
    Point b = {1, 2};
    Point* ptr = &a;

    shift(ptr, 2, -1);
    std::cout << "a: (" << a.x << ", " << a.y << ")" << std::endl;

    ptr = &b;
    ptr->x = a.x + b.y;
    shift(ptr, -3, 5);

    std::cout << "b: (" << b.x << ", " << b.y << ")" << std::endl;
    std::cout << "sum: " << ptr->x + ptr->y << std::endl;

    return 0;
}
```

Solution

Step-by-step trace:

Initial state: `a={3,4}`, `b={1,2}`, `ptr=&a`

`shift(ptr, 2, -1)`: `ptr` points to `a`, so `a.x += 2` → 5, `a.y += -1` → 3. `a={5,3}`

Line 51 prints: `a: (5, 3)`

`ptr = &b`: `ptr` now points to `b`

`ptr->x = a.x + b.y`: `b.x = 5 + 2 = 7`. `b={7,2}`

`shift(ptr, -3, 5)`: `ptr` still points to `b`, so `b.x += -3` → 4, `b.y += 5` → 7. `b={4,7}`

Line 57 prints: `b: (4, 7)`

`ptr->x + ptr->y`: `ptr` still points to `b` → `4 + 7 = 11`

Line 58 prints: `sum: 11`

Complete output:

```
a: (5, 3)
b: (4, 7)
sum: 11
```

Question 2: Tracing — Recursion + Strings

What is the output of the following program? For each recursive call to `weave`, show the value of `i`, the value of `rest`, and the string returned by that call.

```
#include <iostream>
#include <string>

std::string weave(const std::string& s, int i) {
    if (i >= (int)s.size()) return "";
    std::string rest = weave(s, i + 1);
    std::string c(1, s[i]);
    if (i % 2 == 0) return c + rest;
    return rest + c;
}

int main() {
    std::cout << weave("abcde", 0) << std::endl;
    std::cout << weave("xyz", 0) << std::endl;
    return 0;
}
```

Solution

The key insight: the recursive call happens **before** `c` is used. Calls unwind from the deepest first. Even-indexed characters are prepended (`c + rest`); odd-indexed characters are appended (`rest + c`).

Trace of `weave("abcde", 0)`:

Call	`i`	`rest`	Returns
`weave(s, 5)`	5	—	`""`
`weave(s, 4)`	4 (even)	`""`	`"e" + "" = "e"`
`weave(s, 3)`	3 (odd)	`"e"`	`"e" + "d" = "ed"`
`weave(s, 2)`	2 (even)	`"ed"`	`"c" + "ed" = "ced"`
`weave(s, 1)`	1 (odd)	`"ced"`	`"ced" + "b" = "cedb"`
`weave(s, 0)`	0 (even)	`"cedb"`	`"a" + "cedb" = "acedb"`

Trace of `weave("xyz", 0)`:

Call	`i`	`rest`	Returns
`weave(s, 3)`	3	—	`""`
`weave(s, 2)`	2 (even)	`""`	`"z"`
`weave(s, 1)`	1 (odd)	`"z"`	`"z" + "y" = "zy"`
`weave(s, 0)`	0 (even)	`"zy"`	`"x" + "zy" = "xzy"`

Complete output:

```
acedb
xzy
```

Question 3: Debugging — Linked List

The program below intends to reverse a singly linked list in place, print the result, then free all memory. It contains **4 bugs**. For each one, state the line, explain the problem, and write the fix.

```cpp
#include <iostream>

struct Node {
    int data;
    Node* next;
};

Node* reverse(Node* head) {
    Node* prev = head;
    Node* curr = head;

    while (curr != nullptr) {
        curr->next = prev;
        prev = curr;
        curr = curr->next;
    }
    return curr;
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->data << " -> ";
        head = head->next;
    }
    std::cout << "nullptr" << std::endl;
}

void deleteList(Node* head) {
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}

int main() {
    Node* list = nullptr;
    for (int i = 5; i >= 1; i--) {
        Node* n = new Node;
        n->data = i * 10;
        n->next = list;
        list = n;
    }

    std::cout << "Before: ";
    printList(list);

    list = reverse(list);

    std::cout << "After: ";
    printList(list);

    deleteList(list);
    return 0;
}
```

Solution

Bug 1 — Line 104: `prev` initialized to `head` instead of `nullptr`

The last node in the reversed list must point to `nullptr`. By initializing `prev = head`, the first iteration sets `curr->next = head`, making the original head node point to itself and causing an infinite loop.

Fix:

```cpp
Node* prev = nullptr;
```

Bug 2 — Line 108: Forward link overwritten before being saved

`curr->next = prev` destroys the only reference to the next node in the list. After this line, there is no way to advance `curr` forward.

Fix: save the next pointer before modifying `curr->next`:

```
Node* next = curr->next;
curr->next = prev;
```

Bug 3 — Line 110: `curr` advanced using the already-overwritten `next`

After Bug 2′s assignment, `curr->next` points to `prev` (backward), not forward. `curr = curr->next` does not advance to the next node.

Fix: use the saved pointer from Bug 2′s fix:

```
curr = next;
```

Bug 4 — Line 112: Returns `curr` instead of `prev`

After the loop exits, `curr` is `nullptr`. The new head of the reversed list is `prev`.

Fix:

```
return prev;
```

Corrected `reverse`:

```
Node* reverse(Node* head) {
    Node* prev = nullptr;
    Node* curr = head;

    while (curr != nullptr) {
        Node* next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Correct output:

```
Before: 10 -> 20 -> 30 -> 40 -> 50 -> nullptr
After: 50 -> 40 -> 30 -> 20 -> 10 -> nullptr
```

Question 4: Debugging — Binary Search

The program below intends to search a sorted vector for a target value and return its index, or `-1` if not found. It contains **4 bugs**. For each one, state the line, explain the problem, and write the fix.

```
#include <iostream>
#include <vector>

int binarySearch(const std::vector<int>& v, int target) {
    int left = 0;
    int right = v.size();

    while (left < right) {
        int mid = (left + right) / 2;
        if (v[mid] == target) {
            return mid;
        } else if (v[mid] < target) {
            left = mid;
        } else {
            right = mid - 1;
        }
    }
    return 0;
}

int main() {
    std::vector<int> sorted = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};

    std::cout << binarySearch(sorted, 23) << std::endl;   // expected: 5
    std::cout << binarySearch(sorted, 2) << std::endl;    // expected: 0
    std::cout << binarySearch(sorted, 91) << std::endl;   // expected: 9
    std::cout << binarySearch(sorted, 10) << std::endl;   // expected: -1

    return 0;
}
```

Solution

Bug 1 — Line 165: `right = v.size()` should be `right = v.size() - 1`

`v.size()` is one past the last valid index. In the inclusive binary search, `right` should be the index of the last element. Initializing `right` out of bounds can lead to `v[right]` being accessed if the logic allows `mid` to reach it.

Fix:

```
int right = (int)v.size() - 1;
```

Bug 2 — Line 167: `while (left < right)` should be `while (left <= right)`

With `right = v.size() - 1`, using strict `<` skips checking the element when `left == right`. A single remaining candidate element is never examined.

Fix:

```
while (left <= right)
```

Bug 3 — Line 172: `left = mid` should be `left = mid + 1`

When `v[mid] < target`, the target must be strictly to the right of `mid`. Setting `left = mid` can leave `left` stuck at the same value on the next iteration (e.g., when `left + 1 == right`, `mid == left` again), causing an infinite loop.

Fix:

```
left = mid + 1;
```

Bug 4 — Line 177: `return 0` should be `return -1`

When the target is not found, the function is supposed to signal failure with `-1`. Returning `0` is ambiguous since `0` is also the valid index of the first element.

Fix:

```
return -1;
```

Note: `right = mid - 1` on line 174 is correct for the inclusive variant.

Corrected `binarySearch`:

```
int binarySearch(const std::vector<int>& v, int target) {
    int left = 0;
    int right = (int)v.size() - 1;

    while (left <= right) {
```

```
        int mid = (left + right) / 2;
        if (v[mid] == target) {
            return mid;
        } else if (v[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}
```

Correct output:

```
5
0
9
-1
```

Question 5: Programming — Check if Linked List is Sorted

Given the following struct:

```
struct Node {
    int data;
    Node* next;
};
```

Write a recursive function with the following signature:

```
bool isSorted(Node* head)
```

Returns `true` if the list is in non-decreasing order (each element ≤ the next), and `false` otherwise. An empty list and a single-node list are both considered sorted. No loops allowed.

Examples: `1 -> 2 -> 3 -> nullptr` → `true`; `3 -> 1 -> 2 -> nullptr` → `false`; `5 -> 5 -> 5 -> nullptr` → `true`; empty list → `true`; `7 -> nullptr` → `true`

Part A: Pseudocode

Identify your base cases — when can you immediately return `true`? Then describe your recursive case: what condition causes the function to return `false`, and how do you combine the current check with the recursive result?

Solution

```
function isSorted(head):
    // Base case 1: empty list is sorted
    if head == nullptr:
        return true

    // Base case 2: single node is sorted
    if head.next == nullptr:
        return true

    // Recursive case: if current node exceeds the next, order is violated
    if head.data > head.next.data:
        return false

    // Otherwise check the rest of the list
    return isSorted(head.next)
```

Base case 1: An empty list has no elements out of order — return `true`.

Base case 2: A single node has nothing to compare against — return `true`.

Recursive case: If `head->data > head->next->data`, the ordering is already broken — return `false`. Otherwise, the current pair is fine; delegate to `isSorted(head->next)` to check the remainder.

Part B: Implementation

Write the complete function along with a `printList`, `deleteList`, and a `main()` that builds at least two different lists and tests at least 4 cases — include an empty list, a single-element list, a sorted list, and an unsorted list.

Solution

```
#include <iostream>

struct Node {
    int data;
    Node* next;
};

bool isSorted(Node* head) {
    if (head == nullptr) return true;
    if (head->next == nullptr) return true;
    if (head->data > head->next->data) return false;
    return isSorted(head->next);
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->data << " -> ";
        head = head->next;
    }
    std::cout << "nullptr" << std::endl;
}

void deleteList(Node* head) {
```

```
    while (head != nullptr) {
        Node* next = head->next;
        delete head;
        head = next;
    }
}

int main() {
    // Test 1: empty list
    Node* empty = nullptr;
    std::cout << std::boolalpha << isSorted(empty) << std::endl;   // true

    // Test 2: single element
    Node* single = new Node{7, nullptr};
    std::cout << isSorted(single) << std::endl;                    // true
    deleteList(single);

    // Test 3: sorted ascending  1 -> 2 -> 4 -> 4 -> 8
    Node* sorted = new Node{1, new Node{2, new Node{4, new Node{4, new Node{8, nullptr}}}}};
    printList(sorted);
    std::cout << isSorted(sorted) << std::endl;                    // true
    deleteList(sorted);

    // Test 4: unsorted  3 -> 1 -> 2
    Node* unsorted = new Node{3, new Node{1, new Node{2, nullptr}}};
    printList(unsorted);
    std::cout << isSorted(unsorted) << std::endl;                  // false
    deleteList(unsorted);

    return 0;
}
```

Question 6: Programming — Rotating a Vector

Write a function with the following signature:

```
void rotate(std::vector<int>& v, int k)
```

Rotates the vector to the right by `k` positions in place. A right rotation by 1 moves the last element to the front. If `k >= v.size()`, the rotation should still work correctly.

Examples: `{1, 2, 3, 4, 5}` rotated by 2 → `{4, 5, 1, 2, 3}`, `{1, 2, 3}` rotated by 3 → `{1, 2, 3}`, `{1, 2, 3, 4, 5}` rotated by 7 → `{4, 5, 1, 2, 3}`, `{5}` rotated by 4 → `{5}`

Part A: Pseudocode

Explain how you handle the case where `k >= v.size()` using modulo. Identify the index in the original vector where the new first element comes from and describe how you construct the rotated result.

Solution

```
function rotate(v, k):
    if v is empty, return
    k = k mod v.size()     // reduce: rotating by n is the same as rotating by 0
    if k == 0, return      // nothing to do

    // The new first element comes from index (n - k)
    // Build result: last k elements, then the first (n - k) elements
    result = empty vector
    for i from (n - k) to (n - 1):
        append v[i] to result
    for i from 0 to (n - k - 1):
        append v[i] to result

    v = result
```

A right rotation by `k` brings the last `k` elements to the front. After reducing with modulo, the split point is `n - k`: elements `v[n-k]` through `v[n-1]` become the new prefix, and `v[0]` through `v[n-k-1]` become the new suffix.

Part B: Implementation

Write the complete function and a `main()` with at least 3 test cases — include a case where `k = 0`, a case where `k` equals the vector size, and a case where `k` is larger than the vector size.

Solution

```
#include <iostream>
#include <vector>

void rotate(std::vector<int>& v, int k) {
    if (v.empty()) return;
    k = k % (int)v.size();
    if (k == 0) return;

    std::vector<int> result;
    for (int i = (int)v.size() - k; i < (int)v.size(); i++)
        result.push_back(v[i]);
    for (int i = 0; i < (int)v.size() - k; i++)
        result.push_back(v[i]);
    v = result;
}

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

int main() {
    std::vector<int> v1 = {1, 2, 3, 4, 5};
    rotate(v1, 2);
    printVector(v1);   // { 4, 5, 1, 2, 3 }

    std::vector<int> v2 = {1, 2, 3, 4, 5};
    rotate(v2, 0);
    printVector(v2);   // { 1, 2, 3, 4, 5 }  (k=0, no change)
```

```cpp
    std::vector<int> v3 = {1, 2, 3};
    rotate(v3, 3);
    printVector(v3);   // { 1, 2, 3 }  (k == size, full rotation)

    std::vector<int> v4 = {1, 2, 3, 4, 5};
    rotate(v4, 7);
    printVector(v4);   // { 4, 5, 1, 2, 3 }  (7 % 5 = 2)

    return 0;
}
```

Question 7: Programming — Maximum Subarray Sum

Write a function with the following signature:

```
int maxSubarraySum(const std::vector<int>& v)
```

Returns the largest sum of any contiguous subarray of `v`. The subarray must contain at least one element. You may assume `v` is non-empty.

Examples: `{-2, 1, -3, 4, -1, 2, 1, -5, 4}` → `6` (subarray `{4, -1, 2, 1}`), `{1, 2, 3}` → `6`, `{-3, -1, -2}` → `-1`, `{5}` → `5`

Part A: Pseudocode

Describe your approach for checking every possible contiguous subarray. Explain how you track the sum of each subarray you consider and how you update your best result.

Solution

```
function maxSubarraySum(v):
    maxSum = v[0]      // initialize to first element (at least one element required)

    for start = 0 to n-1:
        runningSum = 0
        for end = start to n-1:
            runningSum += v[end]     // extend subarray one element to the right
            if runningSum > maxSum:
                maxSum = runningSum  // new best found

    return maxSum
```

For each possible starting index, we extend the subarray one element at a time, accumulating a running sum. We compare after each extension and update `maxSum` whenever we find a larger total. After all `O(n²)` subarrays have been examined, `maxSum` holds the answer.

Initializing `maxSum` to `v[0]` rather than 0 correctly handles the case where all elements are negative (the answer is the least negative element, not 0).

Part B: Implementation

Write the complete function and a `main()` with at least 4 test cases — include a vector with all negative values, a vector with all positive values, a single-element vector, and a mixed vector where the answer is neither the full array nor a single element.

Solution

```
#include <iostream>
#include <vector>

int maxSubarraySum(const std::vector<int>& v) {
    int maxSum = v[0];
    for (int i = 0; i < (int)v.size(); i++) {
        int sum = 0;
        for (int j = i; j < (int)v.size(); j++) {
            sum += v[j];
            if (sum > maxSum) maxSum = sum;
        }
    }
    return maxSum;
}

int main() {
    // Mixed: answer is subarray {4, -1, 2, 1}
    std::vector<int> v1 = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    std::cout << maxSubarraySum(v1) << std::endl;   // 6

    // All positive: full array is the answer
    std::vector<int> v2 = {1, 2, 3};
    std::cout << maxSubarraySum(v2) << std::endl;   // 6

    // All negative: least negative element is the answer
    std::vector<int> v3 = {-3, -1, -2};
    std::cout << maxSubarraySum(v3) << std::endl;   // -1

    // Single element
    std::vector<int> v4 = {5};
    std::cout << maxSubarraySum(v4) << std::endl;   // 5

    return 0;
}
```