MTH 4300: Algorithms, Computers and Programming II

Spring 2026

Section: SMWA

Midterm 1 (March 4th 2026)

Grading Section (For Instructor Use)

Question	Points	Score
Question 1	5	/ 5
Question 2	5	/ 5
Question 3	7	/ 7
Question 4	10	/ 10
Question 5	10	/ 10
Total	**37**	/ 37

Comments:

Question 1

What is the output of the following program? Trace through step by step, tracking the value of every variable and what each pointer points to after each statement.

```
#include <iostream>

void update(int*& p, int& r) {
    r += *p;
    p = &r;
}

int main() {
    int a = 4, b = 6, c = 10;
    int* ptr = &c;
    int& ref = a;

    std::cout << *ptr << " " << ref << std::endl;

    update(ptr, b);

    std::cout << *ptr << " " << ref << std::endl;
    std::cout << a << " " << b << " " << c << std::endl;

    ref += *ptr;
    *ptr = ref - c;

    std::cout << a << " " << b << " " << c << std::endl;
    std::cout << *ptr << " " << ref << std::endl;

    return 0;
}
```

Solution

Variable tracking:

Initial state: `a = 4`, `b = 6`, `c = 10`, `ptr → c`, `ref = a`

First `cout`: `cout << *ptr << " " << ref` → prints `10 4`

`update(ptr, b)` call: `p` is a reference to `ptr`, `r` is a reference to `b`

- `r += *p` → `b += *ptr` → `b = 6 + 10 = 16`
- `p = &r` → `ptr = &b`

Second `cout`: `cout << *ptr << " " << ref` → `*ptr = b = 16`, `ref = a = 4` → prints `16 4`

Third `cout`: `cout << a << " " << b << " " << c` → prints `4 16 10`

`ref += *ptr`: `a += b` → `a = 4 + 16 = 20`

`*ptr = ref - c`: `b = a - c = 20 - 10 = 10`

Fourth `cout`: `cout << a << " " << b << " " << c` → prints `20 10 10`

Fifth `cout`: `cout << *ptr << " " << ref` → `*ptr = b = 10`, `ref = a = 20` → prints `10 20`

Full output:

```
10 4
16 4
4 16 10
20 10 10
10 20
```

Rubric (5 pts)

1 point per correct output line. Accept minor formatting differences (e.g. extra spaces) if the values are right.

Points	Criteria
1	First `cout` correct: `10 4`
1	Second `cout` correct: `16 4` (requires understanding both `r += *p` and `p = &r`)
1	Third `cout` correct: `4 16 10`
1	Fourth `cout` correct: `20 10 10`
1	Fifth `cout` correct: `10 20`

Question 2

What is the output of the following program? For each recursive call, show the value of `i`, what is printed (if anything), and what value is returned.

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

int f(std::vector<int>& v, int i) {
    if (i == (int)v.size() - 1) return v[i];
    int rest = f(v, i + 1);
    v[i] = rest - v[i];
    std::cout << v[i] << std::endl;
    return v[i] + rest;
}

int main() {
    std::vector<int> v = {2, 5, 3, 4};
    int r = f(v, 0);
    std::cout << r << std::endl;
    std::cout << "Vector: ";
    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;
    return 0;
}
```

Solution

Call stack trace:

- `f(v, 0)` calls `f(v, 1)`
  - `f(v, 1)` calls `f(v, 2)`
    - `f(v, 2)` calls `f(v, 3)`
      - `f(v, 3)`: base case (`i == v.size()-1 = 3`), returns `v[3] = 4`
    - `rest = 4`, `v[2] = 4 - 3 = 1`, prints `1`, returns `1 + 4 = 5`
  - `rest = 5`, `v[1] = 5 - 5 = 0`, prints `0`, returns `0 + 5 = 5`
- `rest = 5`, `v[0] = 5 - 2 = 3`, prints `3`, returns `3 + 5 = 8`

`r = 8`, prints `8`. Vector is now `{3, 0, 1, 4}`.

Full output:

```
1
0
3
8
Vector: 3 0 1 4
```

Rubric (5 pts)

1 point per correct output line. A wrong intermediate value that propagates consistently (e.g. student made one arithmetic error but traced the rest correctly from it) may receive partial credit at instructor discretion.

Points	Criteria
1	`1` printed (from `f(v, 2)`)
1	`0` printed (from `f(v, 1)`)
1	`3` printed (from `f(v, 0)`)
1	`8` printed (return value `r`)
1	Vector line correct: `Vector: 3 0 1 4`

Question 3

The program below intends to build a linked list by appending values one at a time, print the result, then free all memory. It contains **2 bugs**. For each one, state the line, explain the problem, and write the fix.

```
#include <iostream>

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

void append(Node*& head, int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    Node* current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = newNode;
}

void printList(Node* head) {
    while (head->next != 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() {
    Node* list = nullptr;
    append(list, 10);
    append(list, 20);
    append(list, 30);

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

Solution

Bug 1 — in `append`: `while (current->next != nullptr)` dereferences `nullptr` when `head` is `nullptr` (empty list). The first call to `append` with `list = nullptr` sets `current = head = nullptr`, then immediately crashes on `current->next`.

Fix: Add an early return at the start of `append`:

```
if (head == nullptr) {
    head = newNode;
    return;
}
```

Bug 2 — in `printList`: `while (head->next != nullptr)` exits the loop when `head` is the last node, so the last element (`30`) is never printed. The output is `10 -> 20 -> nullptr` instead of `10 -> 20 -> 30 -> nullptr`.

Fix: Change the condition to `head != nullptr` and print each node's data unconditionally:

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

Rubric (7 pts)

3 points per bug, 1 point for correctly identifying exactly 2 bugs (no more, no fewer).

Points	Criteria
1	Bug 1: correctly locates the problem in `append` (dereferencing `nullptr` / empty list crash)
1	Bug 1: explains why it is wrong (traversal starts on a null pointer)
1	Bug 1: provides a correct fix (null check before the loop, assigning `head = newNode`)
1	Bug 2: correctly locates the problem in `printList` (wrong loop condition)
1	Bug 2: explains why it is wrong (last node is never printed)
1	Bug 2: provides a correct fix (change condition to `head != nullptr`)
1	Identifies exactly 2 bugs — no spurious bugs reported

Question 4

Write a recursive function with the following signature:

```
std::string everyOther(const std::string& s, int index)
```

Returns a new string containing only the characters of `s` at even positions (`index` 0, 2, 4, …). No loops allowed.

Examples:

```
everyOther("abcdef", 0) → "ace"
everyOther("hello", 0) → "hlo"
everyOther("x", 0) → "x"
everyOther("", 0) → ""
```

Part A: Pseudocode

```
everyOther(s, index):
  if index >= length of s:
    return ""
  if index is even:
    return s[index] + everyOther(s, index + 1)
  else:
    return everyOther(s, index + 1)
```

Base case: `index` is out of bounds → return empty string.

Recursive case: if `index` is even, include `s[index]` and recurse; otherwise skip and recurse. This works regardless of the starting index.

Part B: Implementation

```
std::string everyOther(const std::string& s, int index) {
    if (index >= (int)s.size()) return "";
    if (index % 2 == 0)
        return std::string(1, s[index]) + everyOther(s, index + 1);
    return everyOther(s, index + 1);
}
```

Rubric (10 pts)

Points	Criteria
Part A — Pseudocode (5 pts)	
1	Correct base case: returns empty when `index` is out of bounds
1	Checks whether `index` is even or odd
1	Includes current character when `index` is even
1	Skips current character when `index` is odd
1	Recurses with `index + 1` in both branches (no loops)
Part B — Implementation (5 pts)	
1	Correct base case (`index >= s.size()`, returns `""`)
1	Uses `index % 2 == 0` (or equivalent) to check parity
1	Correctly includes `s[index]` in the even branch
1	Recursive calls use `index + 1` (not `index + 2`)
1	No loops used anywhere

Question 5

Given the following struct:

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

Write a function with the following signature:

```
Node* mergeSorted(Node* a, Node* b)
```

Both `a` and `b` are the heads of sorted singly linked lists in non-decreasing order. Return the head of a single merged sorted list built by relinking the existing nodes — do not allocate any new nodes.

Examples:

```
1 -> 3 -> 5 and 2 -> 4 -> 6 → 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> nullptr
1 -> 2 -> 3 and nullptr → 1 -> 2 -> 3 -> nullptr
nullptr and nullptr → nullptr
```

Part A: Pseudocode

```
mergeSorted(a, b):
  if a is null: return b
  if b is null: return a
  if a.data <= b.data:
    a.next = mergeSorted(a.next, b)
    return a
  else:
    b.next = mergeSorted(a, b.next)
    return b
```

Base cases: if either list is empty, return the other. Recursive case: pick the smaller head, set its `next` to the merged result of the remaining nodes, and return it.

Part B: Implementation

```
Node* mergeSorted(Node* a, Node* b) {
    if (a == nullptr) return b;
    if (b == nullptr) return a;
    if (a->data <= b->data) {
        a->next = mergeSorted(a->next, b);
        return a;
    } else {
        b->next = mergeSorted(a, b->next);
        return b;
    }
}
```

Alternative solution (iterative)

```
mergeSorted(a, b):
  if a is null: return b
  if b is null: return a
  pick the smaller of a or b as head; advance that pointer
  current = head
  while a is not null and b is not null:
    if a.data <= b.data:
      current.next = a; advance a
    else:
      current.next = b; advance b
    advance current
  attach whichever list is not exhausted to current.next
  return head
```

The iterative approach picks the overall head first, then walks a `current` pointer forward, always attaching the smaller of the two front nodes. After one list is exhausted, the remainder of the other is appended directly.

```
Node* mergeSorted(Node* a, Node* b) {
    if (a == nullptr) return b;
    if (b == nullptr) return a;

    Node* head;
    if (a->data <= b->data) { head = a; a = a->next; }
    else                    { head = b; b = b->next; }

    Node* current = head;
    while (a != nullptr && b != nullptr) {
```

```
        if (a->data <= b->data) { current->next = a; a = a->next; }
        else                    { current->next = b; b = b->next; }
        current = current->next;
    }
    current->next = (a != nullptr) ? a : b;
    return head;
}
```

Rubric (10 pts)

Use whichever rubric matches the student's approach.

Recursive solution:

Points	Criteria
Part A — Pseudocode (5 pts)	
1	Base case: returns `b` when `a` is null
1	Base case: returns `a` when `b` is null
1	Correct comparison to select the smaller head
1	Recurses with the tail of the selected list and the full other list
1	Returns the selected head (no new nodes)
Part B — Implementation (5 pts)	
1	Both null base cases handled correctly
1	Correct comparison (`a->data <= b->data` or equivalent)
1	Correct recursive call in the `a <= b` branch and returns `a`
1	Correct recursive call in the `else` branch and returns `b`
1	No new nodes allocated

Iterative solution:

Points	Criteria
Part A — Pseudocode (5 pts)	
1	Handles the case where either input is null
1	Correctly selects the overall head and advances that pointer
1	Loop continues while both lists are non-empty
1	Attaches the smaller front node each iteration and advances both `current` and that list's pointer
1	Appends the remaining non-empty list after the loop
Part B — Implementation (5 pts)	
1	Both null base cases handled correctly
1	Head selected correctly and its list pointer advanced before the loop
1	Loop condition is `while (a != nullptr && b != nullptr)`
1	Correct comparison, relink, and pointer advance inside the loop
1	Remaining list attached after loop (`current->next = a != nullptr ? a : b`); no new nodes allocated