# Assembly Code Analysis: Problem 10 - Running Maximum Array

## 1. C Code to Ground Truth (gd) ARM Assembly Mapping

### C Code Function:

```c
int *func0(int *numbers, int size) {
    if (size <= 0) {
        return NULL;
    }

    int *out = malloc(size * sizeof(int));
    if (!out) {
        return NULL;
    }

    int max = numbers[0];
    for (int i = 0; i < size; i++) {
        if (numbers[i] > max) max = numbers[i];
        out[i] = max;
    }
    return out;
}
```

### Register Mapping in Ground Truth (gd):

- **x0**: `int *numbers` (input) → `int *out` (return value)
- **w1/x1**: `int size` (array size)
- **x19**: saved `int *numbers` pointer
- **x20**: saved `int size`
- **w21**: saved `int size` (32-bit)
- **w8**: `max` variable (running maximum)
- **x9**: loop counter (decremented)
- **x10**: `out` array pointer (incremented)
- **x11**: `numbers` array pointer (incremented)
- **w12**: temporary for `numbers[i]`

### Line-by-Line C to Assembly Mapping:

#### Size Check:

```c
if (size <= 0) return NULL;
```

**Ground Truth:**

```assembly
cmp w1, #1          ; compare size with 1
b.lt LBB0_5         ; if size < 1, jump to return NULL
```

**Memory Allocation:**

```c
int *out = malloc(size * sizeof(int));
if (!out) return NULL;
```

**Ground Truth:**

```assembly
lsl x0, x21, #2     ; x0 = size * 4 (sizeof(int))
bl _malloc          ; call malloc
cbz x0, LBB0_6      ; if malloc returned NULL, jump to return NULL
```

**Initialize Max:**

```c
int max = numbers[0];
```

**Ground Truth:**

```assembly
ldr w8, [x19]       ; w8 = numbers[0] (max)
str w8, [x0]        ; out[0] = max
```

**Main Loop:**

c

```
for (int i = 0; i < size; i++) {
    if (numbers[i] > max) max = numbers[i];
    out[i] = max;
}
```

### Ground Truth:

assembly

```
LBB0_4:                                 ; Loop start
ldr w12, [x11], #4      ; w12 = numbers[i], increment pointer
cmp w12, w8             ; compare numbers[i] with max
csel w8, w12, w8, gt    ; max = (numbers[i] > max) ? numbers[i] : max
str w8, [x10], #4       ; out[i] = max, increment pointer
subs x9, x9, #1         ; decrement loop counter
b.ne LBB0_4             ; continue if not done
```

### Return:

c

```
return out;
```

### Ground Truth:

assembly

```
; x0 already contains the allocated array pointer
ret
```

## 2. Vertical Comparison: Ground Truth vs Predicted

### Key Differences Found:

| Location | Ground Truth (gd) | Predicted (pred) | Issue |
|---|---|---|---|
| **Label Numbers** | `LBB0_5` (NULL return) <br>`LBB0_6` (exit) | `LBB0_6` (NULL return) <br>`LBB0_7` (exit) | Label numbering differs |
| **Loop Setup** | `sub x9, x21, #1`<br>`add x10, x0, #4`<br>`add x11, x19, #4` | `add x9, x0, #4`<br>`add x10, x19, #4`<br>`sub x11, x21, #1` | **Different variable assignment** |
| **Loop Counter** | Uses `x9` as loop counter (decremented) | Uses `x11` as loop counter (decremented) | Different counter register |
| **Pointer Management** | `x10` = out pointer<br>`x11` = numbers pointer | `x9` = out pointer<br>`x10` = numbers pointer | **Swapped pointer roles** |
| **Critical Addition** | Missing | `mov x0, x8`<br>`b LBB0_7` | **MAJOR ERROR**: Returns max value instead of array pointer! |

## 3. Error Analysis and Root Causes

### Primary Error: Incorrect Return Value

The most critical difference is in the predicted code's block 5:

```assembly
; CORRECT (gd):
; (no extra code - x0 already contains malloc'd array pointer)
b LBB0_6                ; jump to exit, returning array pointer

; INCORRECT (pred):
mov x0, x8              ; ERROR: x0 = max value (w8)
b LBB0_7                ; jump to exit, returning max value!
```

### Critical Issues in Predicted Code:

#### Issue 1: Fundamental Return Value Error

```c
// C code expects: return out;  (pointer to array)
// Ground truth: returns malloc'd array pointer in x0
// Predicted: returns the maximum value instead of array pointer
```

**Impact**: The function returns an integer (max value) cast as a pointer, which will likely cause:

- Segmentation faults when caller tries to access the "array"

- Memory leaks (allocated array is never freed)
- Completely wrong behavior

**Issue 2: Register Role Confusion**

The predicted code swaps pointer roles but maintains correct logic within the loop:

```
assembly

; CORRECT (gd):
add x10, x0, #4          ; x10 = out array pointer (skip first element)
add x11, x19, #4         ; x11 = numbers pointer (skip first element)
; Loop uses: [x11] for input, [x10] for output

; DIFFERENT BUT FUNCTIONAL (pred):
add x9, x0, #4           ; x9 = out array pointer
add x10, x19, #4         ; x10 = numbers pointer
; Loop uses: [x10] for input, [x9] for output
```

This register reassignment is actually **functionally correct** - it's just a different register allocation choice.

**Issue 3: Loop Counter Variable**

```
assembly

; CORRECT (gd):
sub x9, x21, #1          ; x9 = loop counter
subs x9, x9, #1          ; decrement x9 in loop

; DIFFERENT BUT FUNCTIONAL (pred):
sub x11, x21, #1         ; x11 = loop counter
subs x11, x11, #1        ; decrement x11 in loop
```

Again, this is just a different register choice but functionally equivalent.

## Connection to Compiler Optimization:

The predicted code shows signs of **incorrect optimization understanding**:

1. **Register Allocation**: The compiler can freely choose different registers for the same logical operations, which explains the pointer role swaps.
2. **Control Flow Optimization**: The compiler might reorganize basic blocks and labels, explaining the different label numbers.
3. **Critical Misunderstanding**: The predicted code seems to confuse the **return value semantics**. It appears to think the function should return the maximum value rather than the array pointer.

### Root Cause Analysis:

The core issue is a **fundamental misunderstanding of function semantics**:

- **What it should do**: Allocate array, fill with running maximum, return array pointer
- **What pred does**: Allocate array, fill with running maximum, **return the maximum value**

This suggests the prediction model confused this function with a different pattern - perhaps a function that finds and returns the maximum value rather than building a running maximum array.

### Logical Consequences:

1. **Caller Crash**: Any code using the returned "pointer" will likely segfault
2. **Memory Leak**: The allocated array becomes unreachable
3. **Type Confusion**: Integer value interpreted as pointer address
4. **Complete Functional Failure**: The function fails its primary purpose

**Severity**: This is a **critical functional error** that makes the entire function unusable, despite the internal loop logic being mostly correct.