1. Catalan Numbers

(a) Code Explanation:

Main Program:

```
li $v0, 4
    la $a0, prompt_n
    syscall
    li $v0, 5
    syscall
    move $s0, $v0            # store n in $s0

    # Check if n is valid (n >= 0)
    bltz $s0, invalid_input

    # Initialize memory for Catalan numbers with -1 (uncomputed state)
    li $t0, 0
    li $t1, 1000
    li $t2, -1
init_loop:
    beq $t0, $t1, calc
    sll $t3, $t0, 2
    la $t4, catalan
    add $t4, $t4, $t3
    sw $t2, 0($t4)
    addi $t0, $t0, 1
    j init_loop
```

Input Prompt and Syscall:

- First, the code displays a prompt (`prompt_n`) to ask the user for input using the syscall with `v0 = 4` (print string).
- Then it reads the input value `n` using `v0 = 5` (read integer), and the input is stored in the register `$s0`.

Input Validation:

- The input `n` is checked to ensure it's non-negative (`n >= 0`). If `n` is negative, the program jumps to the `invalid_input` section using `bltz` (branch if less than zero).

Catalan Number Initialization:

- The code then initializes a block of memory to store the Catalan numbers. It uses a loop to set 1000 memory locations to `-1`, which acts as an "uncomputed" flag for each Catalan number.
- It works by looping from 0 to 999 (`$t0` tracks the index, `$t1` is 1000) and stores `-1` at each position in the `catalan` array. The addresses are calculated using `sll` (shift left logical) and pointer arithmetic.

Subroutine: Recursive Calculation

```
catalan_recursive:
    # Base case: if n == 0 or n == 1, return 1
    li $t0, 1
    ble $a0, $t0, base_case

    # Check if C(n) is already computed (memoization)
    sll $t1, $a0, 2
    la $t2, catalan
    add $t2, $t2, $t1
    lw $t3, 0($t2)
    li $t4, -1
    bne $t3, $t4, memo_return

    # Recursive case: compute C(n)
    li $v0, 0                   # initialize result to 0
    move $t5, $a0               # preserve n
    li $t0, 0                   # i = 0
```

Base Case (n = 0 or n = 1):

- The code checks if `n` is 0 or 1 (base cases of the Catalan number). If `n` is less than or equal to 1, the Catalan number is 1.
- The base case is handled by comparing the input value `a0` (which contains `n`) with 1 (`t0`). If `n <= 1`, the program jumps to the `base_case` label, where it will return 1.

Memoization Check:

- Before computing C(n)C(n)C(n), the program checks if the value has already been computed and stored.
- The Catalan numbers are stored in an array `catalan`. The address for C(n)C(n)C(n) is computed by shifting `n` left by 2 bits (multiplying by 4) to calculate the memory offset.
- It loads the value of C(n)C(n)C(n) from memory into register `t3`. If the value is `-1`, it means the value hasn't been computed yet, and the program proceeds with the recursive calculation. If it's already computed, it jumps to `memo_return`.

Recursive Case Setup:

- If the value hasn't been computed, the code begins the recursive calculation of C(n). It initializes the result (`v0 = 0`) and prepares to compute the Catalan number using the recursive formula:

$$C(n) = \sum_{i=0}^{n-1} C(i)C(n - i - 1)$$

- The loop counter `t0` is set to 0, and the input `n` is preserved in `t5` to use later in the recursion.

Loop Recursive

```
loop_recursive:
    beq $t0, $t5, store_result
    # Calculate C(i) * C(n-i-1)
    move $a0, $t0
    addi $sp, $sp, -16
    sw $ra, 0($sp)
    sw $t0, 4($sp)
    sw $t5, 8($sp)
    sw $v0, 12($sp)
    jal catalan_recursive
    lw $ra, 0($sp)
    lw $t0, 4($sp)
    lw $t5, 8($sp)
    lw $t6, 12($sp)
    addi $sp, $sp, 16
    move $t1, $v0

    sub $a0, $t5, $t0
    subi $a0, $a0, 1
```

```
    addi $sp, $sp, -20
    sw $ra, 0($sp)
    sw $t0, 4($sp)
    sw $t1, 8($sp)
    sw $t5, 12($sp)
    sw $t6, 16($sp)
    jal catalan_recursive
    lw $ra, 0($sp)
    lw $t0, 4($sp)
    lw $t1, 8($sp)
    lw $t5, 12($sp)
    lw $t6, 16($sp)
    addi $sp, $sp, 20

    mul $t3, $t1, $v0
    add $t6, $t6, $t3
    move $v0, $t6
    addi $t0, $t0, 1
    j loop_recursive
```

Loop Control:

- The loop compares the current value of `i` (stored in `$t0`) with `n` (stored in `$t5`). If `i == n`, the loop is done, and the program jumps to the `store_result` label to store the computed Catalan number.

Compute C(i):

- The value of `i` is copied into `$a0` to prepare for the recursive call to calculate C(i).
- The program stores the current values of registers (`$ra`, `$t0`, `$t5`, and `$v0`) onto the stack to preserve the state before making the recursive call to `catalan_recursive`.
- The `jal catalan_recursive` instruction performs the recursive call, where the result of C(i) is stored in `$v0`.
- After the recursive call, the saved values are restored from the stack, and C(i) is stored in `$t1`.

Multiply and Accumulate:

- After computing both C(i) and C(n−i−1), the program multiplies these values (`$t1` for C(i) and `$v0` for C(n−i−1) and stores the product in `$t3`.
- The result of this multiplication is added to `$t6`, which holds the current sum of the Catalan number calculation.

Update Loop Counter and Repeat:

- The program updates the loop counter (`$t0`) by adding 1 to `i`, and then it jumps back to the beginning of the loop (`loop_recursive`) to calculate the next term of the summation.

Final Result:

- Once all the terms have been computed, the loop ends, and the final Catalan number C(n) is stored in `$v0` and used for further processing

Storing Values

```
store_result:
    sll $t1, $t5, 2
    la $t2, catalan
    add $t2, $t2, $t1
    sw $v0, 0($t2)
    jr $ra
```

- This code block finalizes the recursive computation of C(n)C(n)C(n) by saving the result in memory for future use (memoization).
- It calculates the appropriate memory address for C(n)C(n)C(n), stores the result in the `catalan` array, and then returns control to the calling function using the return address stored in `$ra`.

Taylor Series

(a) Code Explanation:

Main Program:

```
main:
    # Prompt for x
    li $v0, 4                # print string syscall
    la $a0, prompt_x
    syscall
    li $v0, 6                # read float syscall
    syscall
    mov.s $f12, $f0          # x is now in $f12
    # Prompt for n
    li $v0, 4                # print string syscall
    la $a0, prompt_n
    syscall
    li $v0, 5               # read integer syscall
    syscall
    move $t0, $v0           # n is now in $t0
    # Initialize sum to 1.0 (first term in series)
    li $t1, 1               # Initialize $t1 to 1 (for integer 1)
    mtc1 $t1, $f2           # Move integer 1 to floating point register
    cvt.s.w $f2, $f2        # Convert integer 1 to float in $f2 (sum)
    # Loop to calculate the series
    li $t1, 1               # k = 1
```

Initialize the Series Sum:

- The series sum is initialized to `1.0`, which is the first term of the series (commonly seen in expansions like e^x or cosine series).
- An integer `1` is loaded into `$t1`, then moved to the floating-point register `$f2` using `mtc1`. The value is converted to a floating-point number (`cvt.s.w $f2, $f2`), setting the initial sum to `1.0`.

Loop Setup:

- The loop counter `k` is initialized to `1` (`li $t1, 1`), indicating that the calculation will start with the first term after the initial 1.0 term in the series.

Summary:

- The code prompts the user for a floating-point number `x` and an integer `n`, prepares these values for the computation of a series (likely Taylor or similar), and initializes the sum to `1.0`.
- It then sets up the loop counter `k = 1`, which will be used to iterate through the terms of the series expansion.

Subroutine: Calculate K^{th} term:

```
li $t2, 1               # initialize result to 1
    mtc1 $t2, $f0           # move integer 1 to floating point register $f0
    cvt.s.w $f0, $f0        # convert integer 1 to float
    move $t2, $a0           # copy k to $t2 (counter)
power_loop:
    beqz $t2, factorial     # if k == 0, go to factorial part
    mul.s $f0, $f0, $f12    # multiply result by x
    subi $t2, $t2, 1        # decrement k
    j power_loop
# Calculate k!
factorial:
    li $t3, 1               # factorial starts at 1
    move $t2, $a0           # copy k to $t2 (counter)
fact_loop:
    beqz $t2, division      # if k == 0, go to division part
    mul $t3, $t3, $t2       # multiply by the next integer
    subi $t2, $t2, 1        # decrement k
    j fact_loop
```

Power Calculation:

- Computes by x^k multiplying the floating-point register `$f0` (which holds the result) by the floating-point register `$f12` (which holds `x`) in each loop iteration. The loop counter `$t2` decrements with each iteration until it reaches 0.

Factorial Calculation:

- Computes k! by multiplying the integer register `$t3` (which holds the factorial result) by the loop counter `$t2` (which holds `k`) in each loop iteration. The value in `$t2` decreases with each iteration until it reaches 0.

Flow:

- Once x^k(in `$f0`) and k! (in `$t3`) are computed, the program will proceed to the division part (not shown), where it will divide x^k by k! to compute the next term in the series expansion.

Division and Loop

```
loop:
    beq $t1, $t0, done

    mov.s $f12, $f12
    move $a0, $t1
    jal calculate_term

    add.s $f2, $f2, $f0
```

```
    addi $t1, $t1, 1
    j loop

division:
    mtc1 $t3, $f4
    cvt.s.w $f4, $f4
    div.s $f0, $f0, $f4
    jr $ra
```

Main Loop:

- The loop iterates from `0` to `t0-1`, computing each term in the series with the `calculate_term` function. It adds each term (result in `$f0`) to the running total in `$f2`. The loop index `$t1` is incremented each iteration.

Division:

- After the loop, it divides the result of the term calculation (in `$f0`) by the factorial value (converted to floating-point and stored in `$f4`) to get the final term value.

3. Challenges

Catalan Numbers

Challenge 1: Recursive Function Implementation

- **Issue**: Implementing recursion in assembly language can be difficult due to the lack of high-level constructs like function stacks and automatic variable storage.
- **Solution**: To handle recursion, we explicitly managed the stack. The return address (`$ra`) and local variables (like `n` and partial sums) were stored manually on the stack before each recursive call. After the function finished, the stack was carefully restored. By ensuring proper stack management, we avoided overwriting crucial data, which is a common issue in recursive assembly programs.

Challenge 2: Stack Memory Management

- **Issue**: Assembly language requires explicit memory management using the stack for storing variables and return addresses. Without careful stack management, recursion depth could lead to stack corruption
- **Solution**: We reserved space on the stack for local variables (`$a0` for input `n`, and `$ra` for the return address). Each time a recursive call was made, the necessary values were saved, and they were restored once the function returned. Ensuring that the stack pointer (`$sp`) was properly adjusted at the beginning and end of the function call was crucial.

Challenge 3: Looping through Recursive Subproblems

- **Issue**: In the recursive step, the Catalan number calculation involves summing `C(i) * C(n-i-1)` for all `i` from 0 to `n-1`. This required a loop structure within the recursive function, something that's not naturally easy to express in assembly.
- **Solution**: We used a `loop` label to iterate over the values of `i`, calculating the necessary recursive subproblems (`C(i)` and `C(n-i-1)`) during each iteration. Proper indexing and bounds checking were ensured using `bgt` (branch on greater than) to terminate the loop once `i` exceeded `n-1`.

Taylor Series

Challenge 1: Precision Handling with Floating Points

- **Issue**: The Taylor series involves calculations with floating-point numbers, such as powers and divisions. Floating-point arithmetic in MIPS is less precise, and incorrect handling can lead to rounding errors or loss of accuracy.
- **Solution:** Use `cvt.s.w` to correctly convert integers to floating-point values.Ensure floating-point registers are used consistently in calculations to avoid mixing integers and floats.Test the code with both small and large values of `x` and `n` to detect and correct precision issues early on.

Challenge 2: Recursive Function Calls and Stack Management

- **Issue**: Managing recursive function calls requires careful handling of the stack. Forgetting to store return addresses or arguments can lead to crashes or incorrect results.
- **Solution**: Save the return address (`$ra`), arguments, and any temporary registers onto the stack before making recursive calls.Ensure stack space is properly allocated and deallocated for each recursion to prevent stack overflow.

Challenge 4: Accuracy in Division Operations

- **Issue**: Division operations in the Taylor series (`x^k / k!`) can become inaccurate if either the numerator or denominator becomes too large or too small, leading to poor approximation.
- **Solution**: Carefully handle the division by using floating-point division (`div.s`) and ensuring both the numerator (`x^k`) and denominator (`k!`) are stored as floats before the operation.

Program Output

```
Enter the value of n: 5
The nth Catalan number is: 42
-- program is finished running --
```

```
Enter the value of x: 2
Enter the number of terms (n): 10
The approximation of e^x is: 7.388713
-- program is finished running --
```

Real World Scenarios

Embedded Systems and Hardware Programming:

- MIPS programming is widely used in embedded systems, where low-level control of hardware is essential. Assembly programming is crucial for optimizing code to run efficiently on devices with limited resources like microcontrollers, sensors, and other embedded hardware components.
- Understanding how hardware interacts with software at the assembly level enables developers to write more efficient, tailored programs for these systems.

Security and Reverse Engineering:

- In cybersecurity and reverse engineering, knowledge of assembly languages is crucial for understanding vulnerabilities and malware. Security experts analyze compiled binaries, which often involve working with assembly code, to detect vulnerabilities like buffer overflows or malicious instructions.

Teaching and Understanding Computer Architecture:

- Learning MIPS or other assembly languages provides foundational knowledge of how computers process instructions, manage memory, and perform arithmetic at the hardware level. This is valuable in fields like computer architecture, compiler development, and operating systems, where a deep understanding of how code translates to machine operations is needed.