Recursion

Recursion is a programming technique where a function calls itself directly or indirectly. It can be a powerful tool for solving complex problems, but it is important to understand how it works and how to avoid writing recursive functions that can cause stack overflows.

Endless Recursion

The example of endless recursion you provided is a good illustration of what can go wrong when recursion is not used correctly.

The Endless procedure calls itself repeatedly without ever checking for a **base case.** As a result, the stack will continue to grow until it overflows, causing the program to crash.

To rewrite the Endless procedure correctly, we need to add a base case. This is a condition that will cause the procedure to terminate instead of calling itself again.

In the case of the Endless procedure, the base case could be something like "if the input is 0, then return".

Here is a rewritten version of the Endless procedure that includes a base case:

```
; Endless Recursion (Endless.asm)
INCLUDE Irvine32.inc
.data
    endlessStr BYTE "This recursion never stops",0
.code
    main PROC
        call
        Endless
        exit
    main ENDP
    Endless PROC
        mov ecx, 1 ; input parameter
        ; base case
        cmp ecx, 0
        je endless_exit

        ; recursive call
        call Endless

        ; decrement input parameter
        dec ecx
        ; and call again
        jmp Endless

        endless_exit:
        ret
    Endless ENDP
END main
```

This rewritten version of the Endless procedure will now terminate correctly when the input is 0. It will also print the message "This recursion never stops" to the console before it terminates.

When to Use Recursion

Recursion is not a good choice for all problems. It can be inefficient and difficult to debug. However, it can be a powerful tool for solving problems that have repeating patterns. For example, recursion is often used to implement algorithms for traversing linked

lists and trees.

If you are considering using recursion in your program, it is important to make sure that the problem you are trying to solve is a good fit for recursion. You should also carefully design your recursive function to avoid stack overflows.

Pushed on Stack	Value in ECX	Value in EAX
L1	5	0
L2	4	5
L2	3	9
L2	2	12
L2	1	14
L2	0	15

Recursively Calculating a Sum

A recursive procedure is one that calls itself. This can be useful for solving problems that can be broken down into smaller subproblems of the same type.

To calculate the sum of the integers from 1 to n, we can use the following recursive procedure:

```
554 CalcSum(n):
555   if n == 0:
556     return 0
557   else:
558     return n + CalcSum(n - 1)
```

This procedure works by recursively calling itself to calculate the sum of the integers from 1 to n - 1, and then adding n to the result. The base case is when n == 0, in which case the sum is simply 0.

The following table shows a stack trace for the recursive call of CalcSum(5):

Stack Frame	ECX (counter)	EAX (sum)
main()	5	0
CalcSum(5)	4	0
CalcSum(4)	3	4
CalcSum(3)	2	7
CalcSum(2)	1	10
CalcSum(1)	0	11

Explanation of the Table:

The stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed. The stack frame contains the return address, which is the address of the next instruction to be executed after the recursive call returns.

The ECX register contains the counter value for the current recursive call. The EAX register contains the sum of the integers calculated so far.

At the first recursive call to CalcSum(5), the counter value is 4 and the sum is 0. The program calculates the sum of the integers from 1 to 4 by recursively calling CalcSum(4).

At the second recursive call to CalcSum(4), the counter value is 3 and the sum is 0. The program calculates the sum of the integers from 1 to 3 by recursively calling CalcSum(3).

This process continues until the base case is reached, when n == 0.

At this point, the program returns 0 from the recursive call. The program then returns from the recursive call to CalcSum(3), and so on.

By the time the program returns from the recursive call to CalcSum(5), the sum of the integers from 1 to 5 has been calculated and stored in the EAX register. The program can then return the sum from the main() function.

```
;Sum of Integers (RecursiveSum.asm)
INCLUDE Irvine32.inc
.code
main PROC
    mov ecx, 5   ; Set ECX to 5, the number of integers to sum.
    mov eax, 0   ; Initialize EAX to 0; it will hold the sum.
    call CalcSum ; Call the CalcSum function to calculate the sum.
L1:
    call WriteDec ; Display the result in EAX.
    call Crlf     ; Print a new line.
    exit
main ENDP
;-----------------------------------------------
CalcSum PROC
    ; Calculates the sum of a list of integers
    ; Receives: ECX = count
    ; Returns: EAX = sum
    ;-----------------------------------------------
    cmp ecx, 0    ; Compare ECX (counter) with 0.
    jz L2         ; If it's zero, jump to L2 and quit.
    add eax, ecx  ; Add ECX to EAX, updating the sum.
    dec ecx       ; Decrement the counter.
    call CalcSum  ; Recursively call CalcSum to process the next integer.

L2:
    ret
CalcSum ENDP
end main
```

This code first sets up the main procedure, where it initializes ecx to 5 (the number of integers to sum) and eax to 0 (to store the sum). It then calls the CalcSum procedure to calculate the sum. Afterward, it prints the result using WriteDec and adds a new line with Crlf.

The CalcSum procedure is a recursive function that calculates the sum of integers. It checks if ecx (the counter) is zero; if not, it adds the current value of ecx to the sum in eax, decrements ecx, and then makes a recursive call to CalcSum. This process continues until ecx

reaches 0, at which point the function returns (ret).

Factorial of an Integer

The Factorial procedure uses recursion to calculate the factorial of a number. It receives one **stack parameter**, N, which is the number to calculate. The calling program's return address is automatically pushed on the stack by the CALL instruction.

The first thing Factorial does is to push EBP on the stack, to save the base pointer to the calling program's stack. It then sets EBP to the beginning of the current stack frame. This allows the procedure to access its parameters and local variables using base-offset addressing.

Next, Factorial checks the base case, which is when N equals zero. In this case, Factorial returns 1, which is the factorial of 0.

If N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter. This process continues until the base case is reached.

When Factorial returns from a recursive call, it multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

Example Stack Trace:

The following table shows a stack trace for a call to Factorial(3):

Stack Frame	EBP	ESP	N
main()	0x00000000	0x00000004	3
Factorial(3)	0x00000004	0x00000000	3
Factorial(2)	0x00000000	0x00000004	2
Factorial(1)	0x00000004	0x00000000	1
Factorial(0)	0x00000000	0x00000004	0

The stack frame for each recursive call is pushed onto the stack when the CALL instruction is executed. The stack frame contains the return address, which is the address of the next instruction to be executed after the recursive call returns.

The EBP register contains the base pointer to the current stack frame. The ESP register contains the stack pointer, which points to the top of the stack.

The N register contains the value of the parameter passed to Factorial.

At the first recursive call to Factorial(3), the EBP register is set to the beginning of the current stack frame. The N register is loaded with the value 3, which is the parameter passed to Factorial.

Factorial checks the base case, which is when N equals zero. Since N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter.

At the second recursive call to Factorial(2), the EBP register is set to the beginning of the new stack frame. The N register is loaded with the value 2, which is the parameter passed to Factorial.

Factorial checks the base case, which is when N equals zero. Since N is not equal to zero, Factorial recursively calls itself, passing in N - 1 as the parameter.

This process continues until the base case is reached, when N equals zero. At this point, Factorial returns 1, which is the factorial of 0.

The program then returns from the recursive call to Factorial(2). The N register is loaded with the value 2, which is the result of the recursive call.

Factorial multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

The program then returns from the recursive call to Factorial(3). The N register is loaded with the value 6, which is the result of the recursive call.

Factorial multiplies the result of the recursive call by N. This is done because the factorial of N is equal to N multiplied by the factorial of N - 1.

The program then returns to the main() function. The EAX register contains the value 6, which is the factorial of 3.

Let's break down the provided assembly code for calculating the factorial of an integer, explained above, step by step, and I'll explain the key parts in detail.

```
INCLUDE Irvine32.inc            ;Calculating a Factorial (Fact.asm)
.code
main PROC
    push 5                      ;Push the initial value (e.g., 5) on the stack.
    call Factorial              ;Call the Factorial procedure to calculate the factorial.
    call WriteDec               ;Display the result (EAX) on the console.
    call Crlf                   ;Print a new line.
    exit
main ENDP
;-----------------------------------------------
Factorial PROC
    ; Calculates a factorial.
    ; Receives: [ebp+8] = n, the number to calculate
    ; Returns: eax = the factorial of n
    ;-----------------------------------------------
    push ebp            ; Save the current base pointer.
    mov ebp, esp        ; Set up a new base pointer for the current stack frame.
    mov eax, [ebp+8]    ; Get the value of n from the stack.
    cmp eax, 0          ; Check if n is zero.
    ja L1               ; If n is greater than zero, continue; otherwise, go to L2.
    mov eax, 1          ; If n is zero, return 1 as the value of 0!
    jmp L2              ; Jump to the point where we clean up the stack and return.
L1:
    dec eax             ; Decrement n.
    push eax            ; Push the decremented value onto the stack.
    call Factorial      ; Recursively call the Factorial procedure with n-1.
L2:
    pop ebp             ; Clean up the stack by restoring the previous base pointer.
    ret                 ; Return with the result (EAX).
Factorial ENDP
END main
```

Here's an in-depth explanation:

The main procedure begins by pushing the initial value (5 in this case) onto the stack and then calls the Factorial procedure to calculate the factorial.

The Factorial procedure is a recursive function for calculating the factorial of an integer. It first saves the current base pointer on the stack and sets up a new base pointer for the current stack frame.

It retrieves the value of n from the stack (passed as a parameter) into the eax register.

It compares n to 0 using the cmp instruction. If n is greater than 0 (ja - jump above), it proceeds to L1; otherwise, it jumps to L2.

In L1, it decrements n and pushes the new value onto the stack. Then, it makes a recursive call to the Factorial procedure with n-1.

In L2, it pops the base pointer from the stack to clean up the stack frame and returns with the result in EAX.

This recursive approach continues to reduce n until it reaches 0, accumulating the product of each multiplication in EAX.

The result is then returned and displayed in the main procedure.

The program calculates factorials using recursion, and the result for the provided input of 5 would be 120.

Tip:

It is important to keep track of which registers are modified when making recursive calls to a procedure, so that you can save and restore their values if necessary. This is especially important if the register values are needed across recursive procedure calls.

---

1. (True/False): Given the same task to accomplish, a recursive subroutine usually uses more memory than a nonrecursive one. False: Recursive subroutines typically use more memory than nonrecursive subroutines, because they require additional stack space to store the return addresses of the recursive calls.

2. In the Factorial function, what condition terminates the recursion? The recursion terminates when the input parameter, n, is equal to 0.

3. Which instructions in the assembly language Factorial procedure execute after each recursive call has finished? The following instructions in the assembly language Factorial procedure execute after each recursive call has finished:

```
669 mov ebx, [ebp+8]
670 mul ebx
```

These instructions multiply the result of the recursive call by n. This is necessary because the factorial of n is equal to n multiplied by the factorial of n - 1.

What would happen to the Factorial program's output if you tried to calculate 13!? The Factorial program would fail to calculate 13! because the factorial of 13 is too large to be represented in a 32-bit integer.

Challenge: How many bytes of stack space would be used by the Factorial procedure when calculating 5!? The Factorial procedure would use 20 bytes of stack space when calculating 5!. This is because the stack frame for each recursive call requires 4 bytes for the return address and 16 bytes for the local variables.

Here is a breakdown of the stack space requirements:

```
676 Return address: 4 bytes
677 Local variables: 16 bytes
678 Total: 20 bytes
```

The Factorial procedure makes 5 recursive calls when calculating 5!, so the total stack space requirement is 20 bytes per recursive call * 5 recursive calls = 100 bytes.