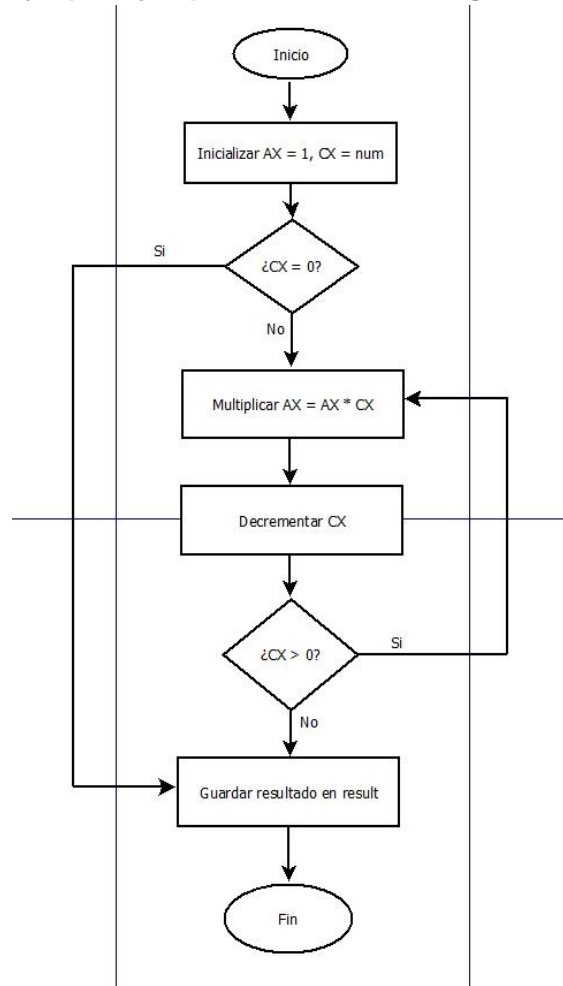


Ejercicio N2: Calcular el factorial de un número.

1- Desarrollar un diagrama de flujo, por ejemplo usando Dia, o alguna versión online.



2- Explicar, comentar las sentencias que no tienen comentarios.

```
; CalcFact.asm - Factorial simple en TASM 16-bit
.MODEL SMALL                ; Modelo de memoria pequeño: código <64K y
datos <64K
.STACK 100h                 ; Reservar 256 bytes para la pila (100h = 256
decimal)

.DATA
    num    DW 5              ; Variable 'num' (word, 16 bits). Número cuyo
factorial calculamos.
    result DW ?              ; Variable 'result' (word, 16 bits).
Guardaremos la parte baja del resultado.

.CODE
START:
    ; Inicializar segmento de datos: DGROUP es el inicio del grupo
de datos por defecto.
    MOV AX, SEG DGROUP      ; AX <- segmento donde están las variables
(DGROUP)
    MOV DS, AX              ; DS <- AX para que las instrucciones que
usan [var] funcionen
```

```

; Inicializar acumulador: AX será el acumulador (parte baja del
producto).
MOV AX, 1 ; AX <- 1 (porque n! se calcula
multiplicando, partimos de 1)

; Cargar contador en CX: LOOP utiliza CX implícitamente.
MOV CX, [num] ; CX <- contenido de 'num'
(direccionamiento directo)

; Si CX = 0 (es decir num = 0), saltar al final: 0! = 1 por
definición.
CMP CX, 0 ; compara CX con 0 (afecta flags)
JE FACT_DONE ; si CX = 0, salta a FACT_DONE (no entra al
bucle)

FACT_LOOP:
; Multiplicación sin signo palabra * palabra:
; MUL r16: hace AX * r16; el producto de 32 bits queda en DX:AX
(DX = high word, AX = low word).
; Aquí multiplicamos el acumulador (AX) por CX.
MUL CX ; DX:AX <- AX * CX

; LOOP decreuenta CX y salta a FACT_LOOP mientras CX != 0.
; Nota: LOOP usa CX internamente (decreuenta y comprueba), por
eso no ponemos DEC CX.
LOOP FACT_LOOP ; CX <- CX - 1 ; if CX != 0 then jump
FACT_LOOP

FACT_DONE:
; Guardar el resultado: almacenamos la palabra baja (AX) en
'result'.
; Si el factorial excede 16 bits, la parte alta está en DX y se
perdería al sólo guardar AX.
MOV [result], AX ; result <- AX (low word del producto)

; Terminar programa y volver a DOS (INT 21h función 4Ch)
MOV AH, 4Ch ; AH = 4Ch (función DOS: terminate program)
INT 21h ; llamar a DOS

END START ; marca el final del módulo y el punto de
entrada START

```

3- Implementar en Turbo Asm y ejecutar.

The screenshot shows the GUI Turbo Assembler 5.1 interface. The main window displays the assembly code for 'CalcFact.asm', which is a 16-bit real-mode program. The code includes comments in Spanish explaining the logic: initializing the accumulator (AX) to 1, loading the counter (CX) from the 'num' variable, and using a LOOP instruction to calculate the factorial. The program uses the MUL instruction for multiplication and the MOV instruction to store the result in the 'result' variable. It also includes a DOS interrupt (INT 21h) to terminate the program. The output window at the bottom shows the execution results, indicating that the program was executed successfully.

```

CalcFact.asm
1 ; CalcFact.asm - Factorial simple en x86 16-bit
2 .MODEL SMALL ; Modelo de memoria pequeño: código <64K y datos <64K
3 .STACK 100h ; Reservar 256 bytes para la pila (100h = 256 decimal)
4
5 .DATA
6 num DW 5 ; Variable 'num' (word, 16 bits). Número cuyo factorial calculamos.
7 result DW ? ; Variable 'result' (word, 16 bits). Guardaremos la parte baja del resultado.
8
9 .CODE
10 START:
11 ; Inicializar segmento de datos: DGROUP es el inicio del grupo de datos por defecto.
12 MOV AX, SEG DGROUP ; AX <- segmento donde están las variables (DGROUP)
13 MOV DS, AX ; DS <- AX para que las instrucciones que usan [var] funcionen
14
15 ; Inicializar acumulador: AX será el acumulador (parte baja del producto).
16 MOV AX, 1 ; AX <- 1 (porque n! se calcula multiplicando, partimos de 1)
17
18 ; Cargar contador en CX: LOOP utiliza CX implícitamente.
19 MOV CX, [num] ; CX <- contenido de 'num' (direccionamiento directo)
20
21 ; Si CX = 0 (es decir num = 0), saltar al final: 0! = 1 por definición.
22 CMP CX, 0 ; compara CX con 0 (afecta flags)
23 JE FACT_DONE ; si CX = 0, salta a FACT_DONE (no entra al bucle)
24
25 FACT_LOOP:
26 ; Multiplicación sin signo palabra * palabra:
27 ; MUL r16: hace AX * r16; el producto de 32 bits queda en DX:AX (DX = high word, AX = low word).
28 ; Aquí multiplicamos el acumulador (AX) por CX.
29 MUL CX ; DX:AX <- AX * CX
30
31 ; LOOP decreuenta CX y salta a FACT_LOOP mientras CX != 0.
32 ; Nota: LOOP usa CX internamente (decreuenta y comprueba), por eso no ponemos DEC CX.
33 LOOP FACT_LOOP
34
35 FACT_DONE:
36 ; Guardar el resultado: almacenamos la palabra baja (AX) en
37 ; 'result'.
38 ; Si el factorial excede 16 bits, la parte alta está en DX y se
39 ; perdería al sólo guardar AX.
40 MOV [result], AX ; result <- AX (low word del producto)
41
42 ; Terminar programa y volver a DOS (INT 21h función 4Ch)
43 MOV AH, 4Ch ; AH = 4Ch (función DOS: terminate program)
44 INT 21h ; llamar a DOS
45
46 END START ; marca el final del módulo y el punto de
47 ; entrada START

```

Output

```

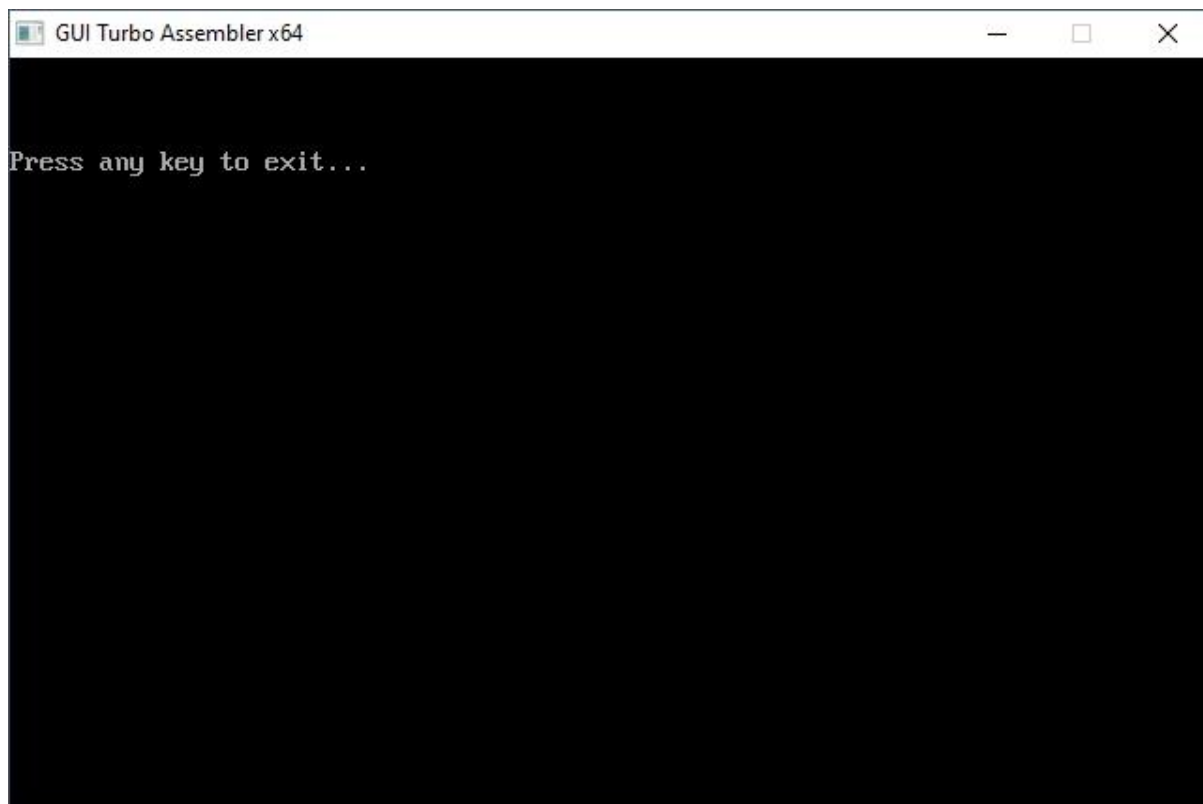
*** Welcome to GUI Turbo Assembler Version 5.1 *** Developed by: Lakshya Jyoti Math (ljmath) | NCA 2015, Tezpur University, IN | Build Time: 23rd November 2023 05:28:03 Rev. 137 ###
[08 oct. 2025 21:47:14.497] Your last session has been successfully restored.
[08 oct. 2025 21:47:14.311] Executing the executable file using the Assembly Launcher. Target file: C:\Users\Gourao\Desktop\CalcFact\CalcFact.exe
[08 oct. 2025 21:47:15.743] The execution of the executable completed successfully.

```

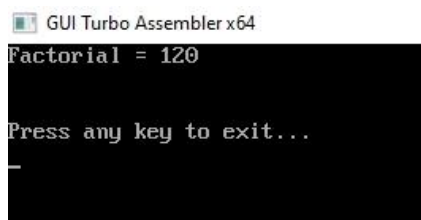
Output Terminal

The execution of the executable completed successfully. Source file: C:\Users\Gourao\Desktop\CalcFact\CalcFact.exe

Turbo 16-bit | Line: 45 Character: 2215

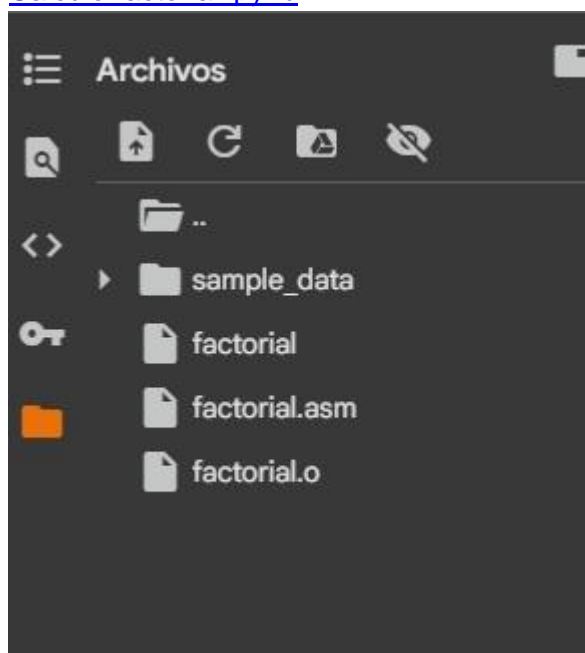


El programa se ejecuta sin problema, aunque no muestra el resultado en pantalla.
Una vez corregido (ver punto 10), se muestra correctamente.



4- Implementar en linux. De no tener uno, usar Google Colab.

[CalculoFactorial.ipynb](#)



```
✓ Ejecutamos

[13]
✓ Os
!./factorial

Factorial = 120
```

5- Pegar las sentencias usadas en linux para compilar y linkear. Lo hizo Uziel.

Instalar NASM:

```
!sudo apt update -y
!sudo apt install nasm y-
```

Crear el archivo con el código:

```
%%writefile factorial.asm
```

Compilar y linkear:

```
!nasm -f elf64 factorial.asm -o factorial.o
!ld factorial.o -o factorial
```

Ejecutar:

```
!./factorial
```

6- Investigar y agregar sentencias para ingresar valores por teclado en un par de ejercicios.

Buscamos en internet y encontramos un artículo en la siguiente página:

<https://webprogramacion.com/lectura-por-teclado-en-el-8086>

Lectura por teclado en el 8086

Microprocesadores

Para poder leer datos por el teclado (entrada Standard) vamos a utilizar la interrupción 21h.

Primeramente necesitamos declarar una variable donde se van a almacenar los caracteres tecleados:

Sintaxis:

```
<nombre_de_la_variable> db <tamaño>?,<tamaño> dup (?),'$'
```

- Ejemplo: bufnum1 db 6,?,6 dup (?),'\$' -> Declaramos un buffer donde podemos almacenar 6 caracteres. Este buffer se va a rellenar con la entrada tecleada.
- Al producirse la lectura, el lugar de la ? (la que está entre los dos 6's) se va a rellenar con el tamaño exacto de la cadena tecleada.

Para llamar a la interrupción para que realice la lectura, tenemos que incluir el siguiente código en nuestro programa (veámoslo sobre el ejemplo anterior):

```
mov ah,0ah
mov dx,offset bufnum1
int 21h
```

Podemos observar en la primera instrucción que la opción de la interrupción 21h la marcamos con 0ah.

Así que probamos agregando el búfer y el bloque de código:

```
bufnum1 db 6, ?, 6 dup(?)
```

```
mov ah, 0Ah
```

```
mov dx, offset bufnum1
```

```
int 21h
```

```
.DATA                                ; Inicio del
num      DW 5
result   DW ?
msg       DB 'Factorial = $'
buffer    DB 6 DUP(0), '$'
```

Aquí reservamos la variable num con el símbolo “?” y agregamos la línea del búfer al final:

```
.DATA                                ; Inicio de
num      DW ?
result   DW ?
msg       DB 'Factorial = $'
buffer    DB 6 DUP(0), '$'
bufnum1   db 6, ?, 6 dup(?)
```

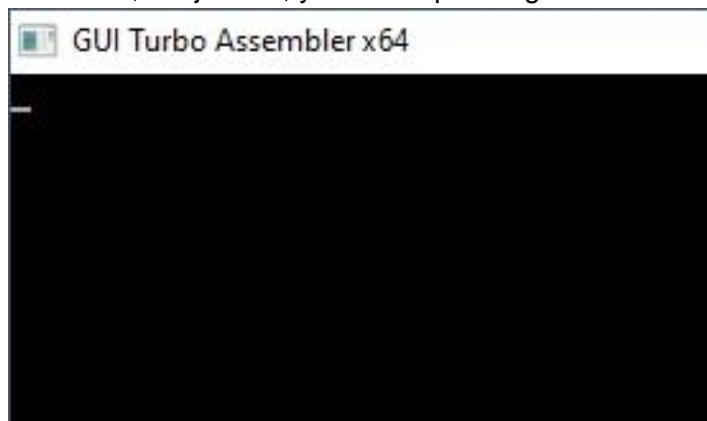
Después agregamos el otro bloque antes del código que calcula el factorial:

```
; --- Leer numero desde tec
mov ah, 0Ah
mov dx, offset bufnum1
int 21h
```

Necesitamos también agregar un conversor, ya que el buffer solo almacena caracteres ascii y no números binarios directamente:

```
mov al, bufnum1+2
sub al, '0'
mov ah, 0
mov [num], ax
```

Entonces, al ejecutar, ya nos da para ingresar el número:



```
GUI Turbo Assembler x64
5
```

```
GUI Turbo Assembler x64
Factorial = 120

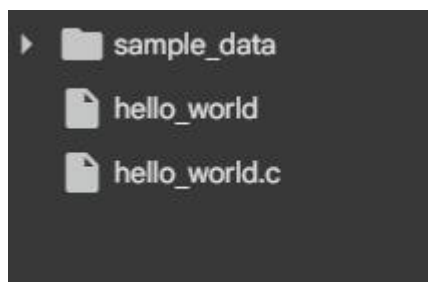
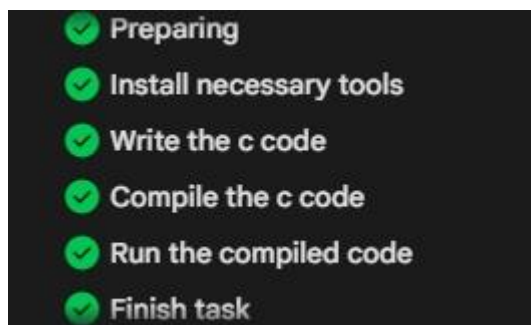
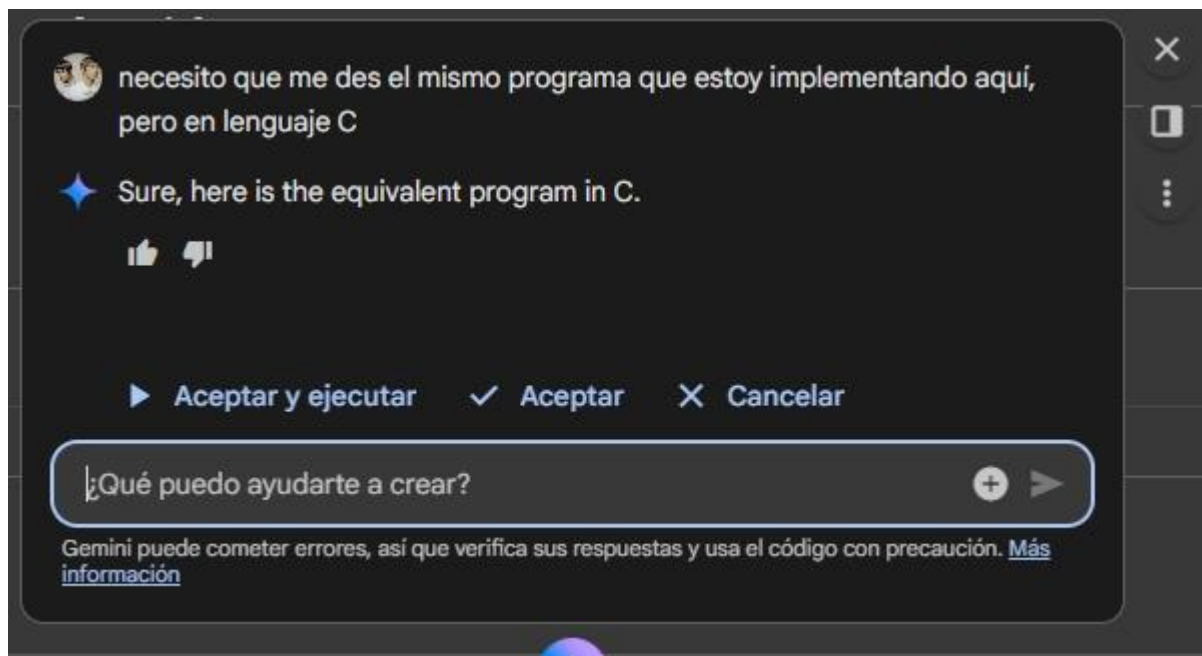
Press any key to exit...
_
```

```
GUI Turbo Assembler x64
?_
```

```
GUI Turbo Assembler x64
Factorial = 5040

Press any key to exit...
_
```

7- Generar en Google Colab, con la IA, la misma versión de problema, pero resuelto en C.
[FactorialC.ipynb](#)



8- Ejecutar y obtener el programa en lenguaje ensamblador del ítem 7.

I need you to create a new cell with the same program, but in assembler language for turbo assembler

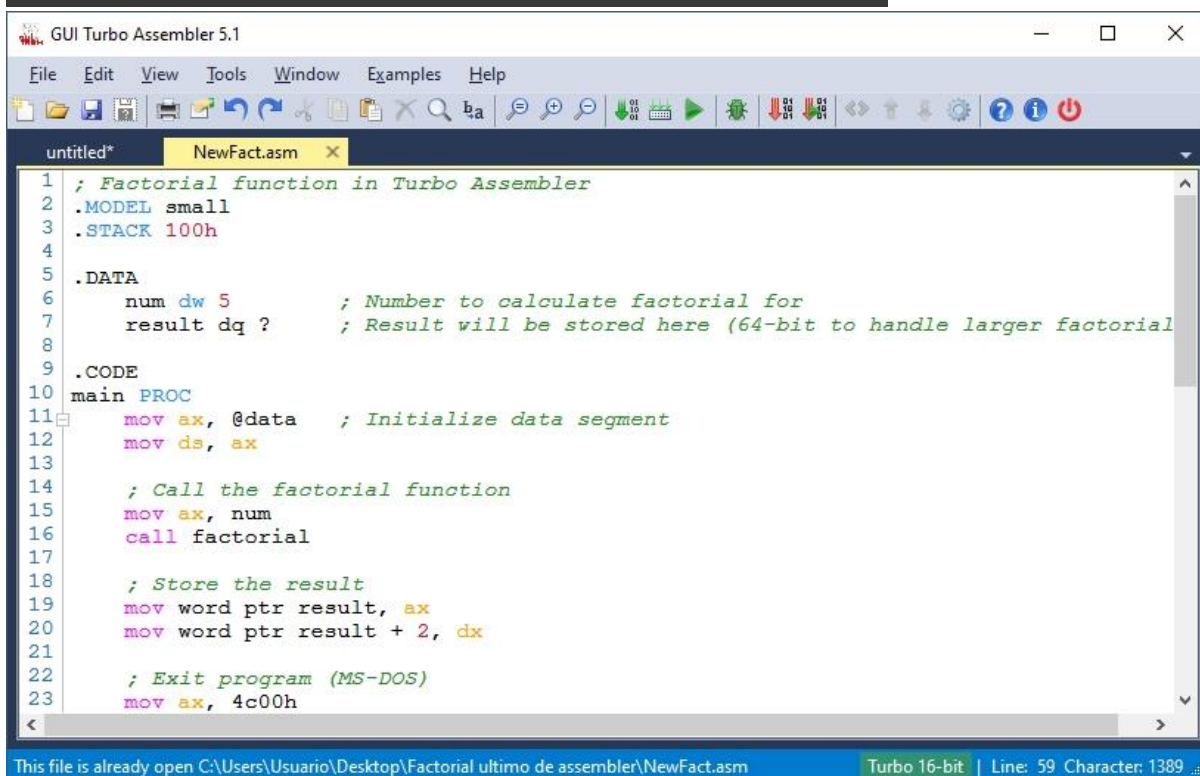
✦ Translating this C program to Turbo Assembler is a complex task. Turbo Assembler is designed for MS-DOS environments and uses a different assembly language syntax than what is typically used or easily runnable in a modern Linux environment like Colab.

While I can generate assembly code, running it directly in Colab would be challenging and likely require setting up a DOS emulator.

Would you still like me to attempt generating the Turbo Assembler code, keeping in mind that running it might be difficult?

just do it, i dont need to run it on colab

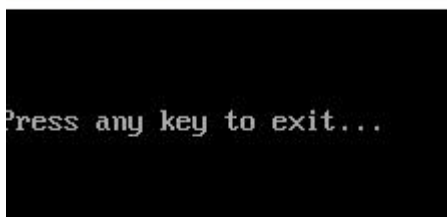
✦ Okay, I understand. Here is the code for the factorial function in Turbo Assembler. Please note that this is a basic translation and might need adjustments depending on the specific Turbo Assembler setup and desired execution environment.



```
GUI Turbo Assembler 5.1
File Edit View Tools Window Examples Help
untitled* NewFact.asm x
1 ; Factorial function in Turbo Assembler
2 .MODEL small
3 .STACK 100h
4
5 .DATA
6     num dw 5          ; Number to calculate factorial for
7     result dq ?       ; Result will be stored here (64-bit to handle larger factorial)
8
9 .CODE
10 main PROC
11     mov ax, @data     ; Initialize data segment
12     mov ds, ax
13
14     ; Call the factorial function
15     mov ax, num
16     call factorial
17
18     ; Store the result
19     mov word ptr result, ax
20     mov word ptr result + 2, dx
21
22     ; Exit program (MS-DOS)
23     mov ax, 4c00h
```

This file is already open C:\Users\Usuario\Desktop\Factorial ultimo de assembler\NewFact.asm Turbo 16-bit | Line: 59 Character: 1389

GUI Turbo Assembler x64



El nuevo código se puede ejecutar, pero, como la primera vez, no muestra resultado alguno.

9- Comparar las versiones de ensamblador.

Vamos a comparar la última versión del código obtenida con la primera versión del código brindada por deepseek.

Código original:

```

1  .MODEL SMALL
2  .STACK 100h
3  .DATA
4      num        DW 5
5      result     DD ?
6  .CODE
7  START:
8      MOV AX, @DATA
9      MOV DS, AX
10
11     MOV AX, 1
12     MOV CX, [num]      ; Direcccionamiento directo
13     CMP CX, 0
14     JE FACT_DONE
15
16 FACT_LOOP:
17     MUL CX              ; AX = AX * CX
18     LOOP FACT_LOOP
19
20 FACT_DONE:
21     MOV [result], AX    ; Almacenar resultado
22
23     MOV AH, 4Ch
24     INT 21h
25 END START

```

Último código:

```

1  ; Factorial function in Turbo Assembler
2  .MODEL small
3  .STACK 100h
4
5  .DATA
6      num dw 5          ; Number to calculate factorial for
7      result dq ?       ; Result will be stored here (64-bit to handle larger factorials)
8
9
10 .CODE
11 main PROC
12     mov ax, @data      ; Initialize data segment
13     mov ds, ax
14
15     ; Call the factorial function
16     mov ax, num
17     call factorial
18
19     ; Store the result
20     mov word ptr result, ax
21     mov word ptr result + 2, dx
22
23     ; Exit program (MS-DOS)
24     mov ax, 4c00h
25     int 21h
26 main ENDP
27
28 ; Factorial function
29 ; Input: AX (number n)
30 ; Output: DX:AX (factorial result)
31 factorial PROC
32     push bp
33     mov bp, sp
34     mov dx, 0           ; Initialize result (high word) to 0
35     mov ax, 1           ; Initialize result (low word) to 1
36
37     cmp word ptr [bp+4], 2 ; Check if n < 2
38     jl factorial_end
39
40     mov cx, word ptr [bp+4] ; Loop counter = n
41     mov bx, 2           ; Start from 2
42
43 factorial_loop:
44     mul dx              ; AX = AX * BX (result * i)
45     push dx             ; Save DX (high word of result)
46     mov dx, 0           ; Clear DX for next multiplication
47     mov ax, bx          ; AX = BX (i)
48     inc ax, bx          ; AX = i + 1
49     mov bx, ax          ; BX = i + 1
50     pop dx              ; Restore DX
51
52     loop factorial_loop
53
54 factorial_end:
55     pop bp
56     ret 2              ; Return and pop the argument
57 factorial ENDP
58
59 END main

```

A simple vista ya podemos ver que el segundo código es mucho más largo que el primero. Ahora vamos por parte.

<pre> .MODEL SMALL .STACK 100h .DATA num DW 5 result DD ? </pre>	<pre> .MODEL small .STACK 100h .DATA num dw 5 result dq ? </pre>
---	---

El primer bloque es casi igual, salvo por el resultado, que en el original es dd (32 bits) y en el último es dq (64 bits).

<pre> .CODE main PROC mov ax, @data mov ds, ax ; Call the factc mov ax, num call factorial </pre>	<pre> .CODE START: MOV AX, @DATA MOV DS, AX MOV AX, 1 MOV CX, [num] CMP CX, 0 JE FACT_DONE </pre>
--	--

En el segundo bloque ya cambia bastante. Ambos inicializan DS con @DATA. y ambos pretenden usar AX/num como entrada al cálculo.

El original carga CX con [num] y luego ejecuta el bucle local; no hay llamada a función.

El último código usa mov ax, num y luego call factorial, dando a entender que la función esperaba el argumento en pila o en ax, pero la función usa [bp+4] para leer el argumento (es decir, espera argumento en la pila).

```
FACT_LOOP:
    MUL CX
    LOOP FACT_LOOP

; Init
mov dx, 0
mov ax, 1

cmp word ptr [bp+4], 2
jl factorial_end

mov cx, word ptr [bp+4]
mov bx, 2 ; Start

factorial_loop:
    mul bx ; AX =
    push dx ; Save
    mov dx, 0 ; Clear
    mov ax, bx ; AX =
    inc ax ; AX =
    mov bx, ax ; BX =
    pop dx ; Restore
    loop factorial_loop
```

El código de loop del factorial del primer código es corto, conciso y eficiente. Empieza con AX=1, CX=num, y la primera iteración multiplica por CX, luego LOOP decrementa CX y salta si no es 0. El código nuevo es extenso y complejo de forma innecesaria. La función pretende iterar multiplicando por BX que se incrementa (2,3,4,...). Esa es otra forma válida de construir, pero tiene errores: mul bx usa AX * BX -> DX:AX. Pero justo después hacen push dx y luego mov dx,0; mov ax, bx; inc ax; mov bx, ax; pop dx. Esa secuencia no conserva correctamente DX:AX: sobrescriben AX con BX (perdiendo el resultado) y simplemente restauran DX. Luego loop se repite, pero AX ya no contiene el producto acumulado. En otras palabras: pierden el acumulador. Por otra parte la función espera su argumento en [bp+4], pero el call no le pasó el argumento. Además, ret 2 al final intenta limpiar 2 bytes de argumento de la pila, pero como no hubo push del argumento, esto corrompe la pila. Resultado: el bucle no mantiene el producto; el código es incorrecto y no funciona como es esperado.

```
FACT_DONE:
    MOV [result], AX
    MOV AH, 4Ch
    INT 21h
END START

; Store the result
mov word ptr result, ax
mov word ptr result + 2, dx

; Exit program (MS-DOS)
mov ax, 4c00h
int 21h
```

El código original guarda solo AX en result (32-bit result DD ? queda sólo parcialmente llenado), y termina con INT 21h AH=4Ch.

El último código intenta almacenar DX:AX en los primeros 4 bytes de result (como 32 bits). Si la intención era guardar 64 bits (dq), falta guardar las 4 bytes altas. Salida por INT 21h.

En conclusión, el código original es simple, directo y funcional. Calcula el factorial mediante un bucle iterativo usando registros básicos (AX y CX) y operaciones simples de multiplicación (MUL CX). Está escrito en estilo de MASM/TASM para DOS, sin funciones ni parámetros. Es eficiente y fácil de entender, aunque limitado a números pequeños (16 bits).

En cambio, el código convertido se complejiza, introduciendo una función (PROC factorial), una convención de llamada por pila, y una variable result más grande (DQ) para soportar valores mayores. Sin embargo, presenta varios errores lógicos y estructurales.

En resumen, el código original es correcto pero básico, mientras que el convertido es más elaborado en apariencia, aunque incorrecto en la práctica.

10- Observar situaciones problemáticas e identificar metodología de resolución de problemas.

Problema #1

Al intentar ejecutar el código en turbo assembler vimos que no mostraba resultado. Al realizar el cálculo, la ventana de comandos solo mostraba “Press any key to continue”.

Nos planteamos: ¿Por qué no se muestra el resultado?

Podríamos deducir que una parte del código impide que se muestre el resultado, ya que al ejecutarlo no da errores.

Decidimos consultar con una inteligencia artificial para que analice el código de forma más eficiente y nos diga si en efecto es así y/o hay un error. Si hay un error, también lo dirá.

Hacemos la consulta y vemos el resultado:

Mirá tu parte final:

asm

Copiar código

FACT_DONE:
MOV [result], AX ; Almacenar resultado
MOV AH, 4Ch
INT 21h

♦ MOV [result], AX → guarda el resultado en la variable result .

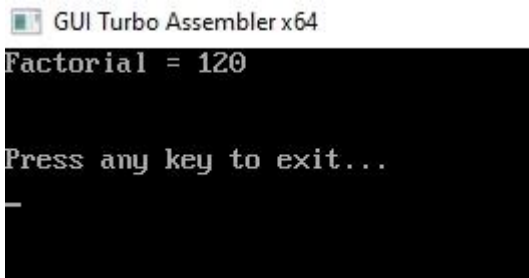
♦ MOV AH, 4Ch + INT 21h → le dice a DOS “terminar el programa”.

👉 O sea: el programa nunca intenta mostrar nada, solo guarda el resultado en memoria y termina.

El chat confirma nuestras sospechas y nos dice precisamente lo que está mal. También nos dice cómo corregirlo y qué es necesario cambiar para ello. Principalmente nos dice de agregar estas líneas:

Explicación de lo nuevo	
Sección	Qué hace
msg DB 'Factorial = \$'	Texto que se mostrará antes del número. El signo \$ indica fin de cadena para DOS.
buffer DB 6 DUP(?)	Espacio donde convertimos el número a texto.

Tomamos el código con los cambios y lo ejecutamos:



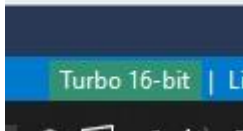
El resultado es 120 porque el número que toma (5) ya está definido en las primeras líneas del código.

```
.DATA                                ; Inicio
num      DW 5
result   DW ?
msg       DB 'Factorial = $'
buffer    DB 6 DUP(0), '$'
```

Problema #2

Al intentar implementar el código en google colab siguiendo el punto 4, vimos que al intentar compilar, simplemente no se podía. Nos da errores al compilar, y nos da error por comando inválido, porque no crea el ejecutable ni el objeto.

Al observar turbo assembler vemos:



Nos preguntamos ¿Por qué el programa no funciona?

Pensamos una posible hipótesis: al igual que en la actividad anterior hubo un problema de que el código estaba en una arquitectura diferente. Aquí podría estar pasando lo mismo.

Pensamos que, si es así, al convertir el código a la arquitectura adecuada, por ejemplo 64 bits, va a funcionar perfectamente.

Consultamos con el chatbot y nos confirma que “el código original está diseñado para **MASM/DOS de 16 bits**, y Google Colab no es un entorno DOS, sino **Linux moderno de 64 bits**”

Hay ciertas instrucciones e interrupciones que no son compatibles con linux y que hay que cambiar. Por ejemplo:

Reemplazar	Por
.MODEL SMALL	section .data .text
AH=09h	write (rax=1)
AH=4Ch	exit (rax=60)
AX, CX, DX	RAX, RCX, RDX

Entonces cambiamos las instrucciones y ejecutamos el código en Colaboratory. Como vimos en el punto 4, funciona.

11- Crear un repositorio de Github y subir todos los archivos. En dicho repositorio deben ser colaboradores todos los miembros del equipo. Repartir tareas, hacer commits y push/pull.

Objetivo a imitar: <https://github.com/OpenWaste/Open-Waste>