Optimización de Código en Compiladores

¿Qué es Optimizar?

El Cheff y la Receta

Actividad

1. Imagina que tu programa es una receta de cocina

   - Ejemplo: "Hacer un sándwich":

     ```python
     pan = "integral"
     queso = "cheddar"
     print(pan + queso + pan)  # Resultado: "integralcheddarintegral"
     ```

2. Optimizar es como un chef que:

   - Elimina pasos innecesarios (¿por qué cortar el pan dos veces si ya viene en rebanadas?).
   - Reorganiza la cocina (tener el queso cerca para no caminar 10 pasos).
   - Usa herramientas más rápidas (un cuchillo afilado vs. uno oxidado).

Contesta lo siguiente:

- ¿Qué pasos de tu vida diaria podrías "optimizar"? (Ej: camino a la escuela, tareas).

Ejemplo Real: Código No Optimizado vs Optimizado

Caso: Calcular el cuadrado de un número 1,000 veces.

-Código en C (no optimizado)

```c
#include <stdio.h>

int main() {
    int suma = 0;
    for (int i = 0; i < 1000; i++) {
        suma += i * i;  // Suma los cuadrados del 0 al 999
    }
    printf("Suma: %d\n", suma);
    return 0;
}
```

1. Compilando (sin optimizar):

- El compilador **no hace ningún truco**.
- Genera un bucle que calcula i * i **1000 veces** (aunque el resultado se podría precalcular).
- El programa será **lento**, pero fácil de depurar.

Código en bajo nivel (no optimizado)

código ensamblador x86-64 implementa una función main() que calcula la **suma de los cuadrados de los números del 0 al 998** y luego imprime el resultado con printf

```
.LC0:
        .string "Suma: %d\n"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 0
        jmp     .L2
.L3:
        mov     eax, DWORD PTR [rbp-8]
        imul    eax, eax
        add     DWORD PTR [rbp-4], eax
        add     DWORD PTR [rbp-8], 1
.L2:
        cmp     DWORD PTR [rbp-8], 999
        jle     .L3
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

.LC0:
  .string "Suma: %d\n" ; Define una cadena de texto para imprimir en printf

main:
  push   rbp   ; Guarda el valor actual de rbp en la pila
  mov   rbp, rsp   ; Establece un nuevo marco de pila
  sub   rsp, 16   ; Reserva espacio en la pila para variables locales

  mov   DWORD PTR [rbp-4], 0 ; Inicializa la variable acumuladora en 0
  mov   DWORD PTR [rbp-8], 0 ; Inicializa el contador en 0
  jmp   .L2   ; Salta a la verificación del bucle

.L3:

```
  mov    eax, DWORD PTR [rbp-8]  ; Carga el valor del contador en eax
  imul   eax, eax            ; Calcula el cuadrado del contador (eax * eax)
  add    DWORD PTR [rbp-4], eax   ; Suma el cuadrado al acumulador
  add    DWORD PTR [rbp-8], 1     ; Incrementa el contador en 1

.L2:
  cmp    DWORD PTR [rbp-8], 999   ; Compara el contador con 999
  jle    .L3                ; Si es menor o igual, repite el bucle

  mov    eax, DWORD PTR [rbp-4]   ; Carga el resultado final en eax
  mov    esi, eax             ; Mueve el resultado a esi (argumento para printf)
  mov    edi, OFFSET FLAT:.LC0    ; Carga la dirección de la cadena "Suma: %d\n" en edi
  mov    eax, 0               ; Limpia eax antes de llamar a printf
  call   printf               ; Llama a printf para imprimir el resultado

  mov    eax, 0  ; Retorna 0 (indicando éxito en la ejecución)
  leave        ; Restaura el marco de pila
  ret          ; Finaliza la ejecución de main
```

Resumen de cada sección:

- **Definición de** .LC0: Se usa para almacenar el texto de salida.
- **Inicio de** main(): Se prepara el marco de pila y se inicializan variables (suma y contador).
- **Bucle** .L3: Calcula el cuadrado de cada número y lo suma a la variable acumuladora.
- **Verificación en** .L2: Compara el contador con 999 para determinar si sigue el bucle.
- **Impresión con** printf: Muestra el resultado final en pantalla.
- **Finalización**: Se limpia la pila y se retorna 0.

CÓDIGO OPTIMIZADO

El compilador dice: *"Esto es una suma de cuadrados! ¡Puedo usar una fórmula matemática!"*

- Reemplaza el bucle con algo como:

```
suma = 332833500;  // Resultado directo de la fórmula n(n+1)(2n+1)/6 (para n=999)
```

¿Qué hizo el compilador?

1. **Eliminó el bucle**: Detectó que el cálculo era una **suma constante** (de 0^2 a 999^2) y lo reemplazó con la fórmula matemática:

$$\text{Suma} = \frac{n(n+1)(2n+1)}{6} \quad (\text{donde } n = 999)$$

2. **Precalculó el resultado**: En tiempo de compilación, resolvió `(999 * 1000 * 1999) / 6 = 332833500`.
3. **Generó código directo**: El programa ahora simplemente asigna el valor precalculado a `suma`.

Código de bajo nivel optimizado

```
1   .LC0:
2           .string "Suma: %d\n"
3   main:
4           sub     rsp, 8
5           mov     esi, 332833500
6           mov     edi, OFFSET FLAT:.LC0
7           xor     eax, eax
8           call    printf
9           xor     eax, eax
10          add     rsp, 8
11          ret
```

Donde:

.LC0:
  .string "Suma: %d\n" ; Define la cadena de formato para printf
main:
  sub rsp, 8 ; Reserva 8 bytes en la pila para alineación o variables temporales
  mov esi, 332833500 ; Carga el valor 332833500 en el registro `esi` (segundo argumento de printf)
  mov edi, OFFSET FLAT:.LC0 ; Carga la dirección de la cadena "Suma: %d\n" en `edi` (primer argumento de printf)
  xor eax, eax ; Limpia `eax` antes de la llamada a printf (convención de llamadas en x86-64)
  call printf ; Llama a printf para imprimir "Suma: 332833500"
  xor eax, eax ; Limpia `eax` antes de salir
  add rsp, 8 ; Restaura el puntero de pila a su estado original
  ret ; Finaliza la ejecución de main

CONCLUSIONES

- El compilador **no altera el código fuente**, pero su versión optimizada **se comporta como si** se hubiera escrito de la forma más eficiente.
- Las optimizaciones dependen del contexto: matemáticas, bucles, funciones, etc.

Contesta las siguientes preguntas

1. **¿Por qué el compilador reemplazó el bucle** for **por una fórmula matemática?**
2. **¿Qué ventajas tiene calcular** (999 * 1000 * 1999) / 6 **en lugar de sumar** i * i **1000 veces?**
3. **¿En qué casos el compilador no podría aplicar esta optimización?**