



PROGRAMACIÓN EFICIENTE
INGENIERÍA EN INFORMÁTICA

2023

Profesor: **Maximiliano A. Eschoyez**
Alumnos: **Calcara Juan Pablo - Millia Jorge**
Fecha: **22/02/23**

Resumen

El objetivo de este Trabajo Final es realizar mejoras a un software para obtener un producto mejor que el original. Para ello se deberán detectar problemáticas de algún software elegido por los estudiantes y aplicarlas.

Consigna

El objetivo de este Trabajo Final es tomar un programa de alguna asignatura anterior, proyecto personal o descargado de Internet y analizarlo en detalle suficiente como para poder determinar qué correcciones, mejoras y optimizaciones se deberían aplicar.

Se deberán analizar los siguientes puntos:

- Cobertura del código fuente.
- Detección de las funciones que realizan el mayor consumo de CPU.
- Redundancia de operaciones aritméticas–lógicas.
- Modo de utilización de la memoria asignada en forma dinámica.
- Modo de acceso a la memoria caché y principal.

Las pruebas a realizar deberán estar sistematizadas para así tener una medición objetiva y repetible. Una vez analizado el software, se deberán proponer mejoras a realizarle.

Luego, se procederá a su implementación.

Finalmente, se deberá hacer un estudio comparativo entre el software original y su versión mejorada aplicando, en caso de ser posible, las mismas pruebas de software sistematizadas. En esta etapa se incluirán todas las tablas y gráficos necesarios para mostrar el impacto de los cambios realizados.

Presentación del Trabajo Final

Código Fuente

El código fuente del programa original y el optimizado deberán entregarse a través del enlace correspondiente en la plataforma MiUBP del examen final (<http://mi.ubp.edu.ar/>). En dicho enlace se deberá subir un único archivo en formato ZIP conteniendo todos los código fuente que se requieran para la realización del trabajo práctico.

Informe Escrito

Se entregará al profesor un informe escrito (solo versión digital en PDF) donde se debe describir la problemática abordada en el trabajo final, el desarrollo de la solución propuesta y una conclusión. El texto deberá ser conciso y con descripciones apropiadas. No se debe incluir el código fuente, sino los textos necesarios para realizar las explicaciones pertinentes.

INTRODUCCIÓN

Se realizaron las pruebas sobre el programa original ([CodigoSinOptimizar.c](#)) y se compararon contra el programa optimizado ([CodigoOptimizado.c](#))

Para ello se utilizaron las siguientes herramientas:

- **INDENT** → para mejorar el aspecto del código.
- **GCOV** → para analizar el uso del código
- **GPROF** → para estudiar el desempeño del código, funciones y llamadas.
- **VALGRIND** → para observar las llamadas del software al sistema.

Esto implica que al verificar el programa sea cumpla con lo siguiente:

- Cumple con su propósito.
- Libera memoria de manera correcta.
- Utiliza la cantidad de memoria apropiada.
- Es eficiente en cuanto a tiempo de ejecución y acceso a memoria.
- Llama a las funciones solo las veces que se necesita.,
- No exista código muerto.

DESARROLLO

INDENT

El programa original no presentaba una correcta indentación y eso dificultaba la lectura del mismo, se utilizó esta herramienta para corregir este problema

Se ejecuta el siguiente comando, para corregir los espacios en blancos y la indentación del código fuente.

- [ORIGINAL → CodigoSinOptimizar.c](#)

```
~/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/ORIGINAL$ indent CodigoSinOptimizar.c -o CodigoSinOptimizar.c -kr
~/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/ORIGINAL$
```

- [OPTIMIZADO → CodigoOptimizado.c](#)

```
/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/OPTIMIZADO$ indent CodigoOptimizado.c -o CodigoOptimizado.c -kr
/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/OPTIMIZADO$
```

Conclusión: como resultado ya tenemos el código más legible

GCOV

La herramienta 'gcov' convierte los archivos de cobertura sin procesar (.gcda y .gcno) en archivos .gcov que luego son procesados por gcovr. Los archivos gcno son generados por el compilador. Los archivos gcda se generan cuando se ejecuta el programa instrumentado.

- [ORIGINAL → CodigoSinOptimizar.c](#)

Ejecutamos el comando **g++ -fprofile-arcs -ftest-coverage CodigoSinOptimizar.c**

```
~/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/ORIGINAL$ g++ -fprofile-arcs -ftest-coverage CodigoSinOptimizar.c
~/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/ORIGINAL$
```

Se generan dos archivos:


a.out

a-CodigoSinOptimizar.gcno

Ejecutamos el comando **./a.out** y genera el archivo **a-CodigoSinOptimizar.gcda**, por último ejecutamos **gcovr -r. --html --html-details -o reporte.html**

```
FINAL PEF 23/Pruebas en Ubuntu/ORIGINAL$ gcovr -r. --html --html-details -o reporte.html
FINAL PEF 23/Pruebas en Ubuntu/ORIGINAL$
```

Se genera un archivo html con la información gcov


GCC Code Coverage Report					
Directory: .		Exec		Total	Coverage
Date: 2023-02-21 20:26:09		Lines:	207	212	97.6%
Legend: low: >= 0% medium: >= 75.0% high: >= 90.0%		Branches:	153	209	73.2%
File	Lines		Branches		
CodigoSinOptimizar.c			97.6%	207 / 212	73.2% 153 / 209

Aca podemos observar que hay código que no es utilizado, por lo tanto se identifica como código muerto identificando las líneas con los caracteres #####

328		20002	double f = 1;	
329		20002	if (n == 0)	
330	▶ 1/2		f = 0;	
331				
332				
333				
334				
335	▶ 2/2	20002	else if (n == 1)	
336				
337	▶ 2/2	1000000000	for (int i = 2; i <= n; i++)	
338				
339		999999999	f = 1;	
340		20002	return f;	
341				
342				
343				
344				
345				
346		10001	int fibonacci(int n)	
347			{	
348				
349		10001	int f = 1, anterior = 1, actual = 1;	
350				
351	▶ 1/2	10001	if (n == 0)	
352				
353				
354				
355	▶ 2/2	10001	else if (n == 1)	
356				

● OPTIMIZADO → [CodigoOptimizado.c](#)

Ejecutamos el comando `gcc -fprofile-arcs -ftest-coverage CodigoOptimizado.c`

GCC Code Coverage Report					
Directory: .		Exec		Total	Coverage
Date: 2023-02-21 20:50:08		Lines:	209	213	98.1%
Legend: low: >= 0% medium: >= 75.0% high: >= 90.0%		Branches:	80	84	95.2%
File	Lines		Branches		
CodigoOptimizado.c			98.1%	209 / 213	95.2% 80 / 84

Generated by GCOV (Version 5.0)

Conclusión: Se puede observar que el programa optimizado ya no muestra el código muerto y tampoco los warnings del original.

GPROF

Ahora realizamos un análisis de comportamiento y rendimiento del código.

- Compilamos y linkeamos el programa usando la bandera (pg)

```
$ g++ -pg CodigoSinOptimizar.c
```

```
$ g++ -pg Código Optimizado.c
```

- Luego ejecutamos

```
$ ./a.out
```

- Utilizamos el gprof para analizar el archivo generado anteriormente y generar la información requerida.

```
$ gprof ./a.out
```

- [ORIGINAL](#) → [CodigoSinOptimizar.c](#)

FLAT PROFILE

```
1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4 % cumulative self self total
5 time seconds seconds calls ms/call ms/call name
6 63.16 0.24 0.24 20002 0.01 0.01 Factorial(int)
7 23.68 0.33 0.09 10001 0.01 0.01 Fibonacci(int)
8 5.26 0.35 0.02 4 5.00 5.00 preOrderTraversal(node*)
9 2.63 0.36 0.01 200000 0.00 0.00 insertNode(node*, int)
10 2.63 0.37 0.01 4 2.50 2.50 postOrderTraversal(node*)
11 2.63 0.38 0.01 main
12 0.00 0.38 0.00 49101 0.00 0.00 createNode(int)
13 0.00 0.38 0.00 10000 0.00 0.00 __gnu_cxx::__promote_2<int, int,
__gnu_cxx::__promote<int, std::__is_integer<int>::__value>::__type, __gnu_cxx::__promote<int,
std::__is_integer<int>::__value>::__type>::__type std::pow<int, int>(int, int)
14 0.00 0.38 0.00 4 0.00 0.00 inOrderTraversal(node*)
15 0.00 0.38 0.00 3 0.00 0.00 OrMatriz(int)
16 0.00 0.38 0.00 1 0.00 0.00 sumar()
17 0.00 0.38 0.00 1 0.00 0.00 ProductoX(int)
18
```

CALL GRAPH (EXPLANATION FOLLOWS)

```

53
54      Call graph (explanation follows)
55
56
57 granularity: each sample hit covers 4 byte(s) for 2.63% of 0.38 seconds
58
59 index % time    self  children   called    name
60
61 [1]    100.0    0.01    0.37          <spontaneous>
62          0.24    0.00    20002/20002    main [1]
63          0.09    0.00    10001/10001    Factorial(int) [2]
64          0.02    0.00         4/4    Fibonacci(int) [3]
65          0.01    0.00    200000/200000    preOrderTraversal(node*) [4]
66          0.01    0.00         4/4    insertNode(node*, int) [5]
67          0.00    0.00    10000/10000    postOrderTraversal(node*) [6]
68          __gnu_cxx::__promote<int, std::__is_integer<int>::__value>::__type, __gnu_cxx::__promote<int,
69          std::__is_integer<int>::__value>::__type>::__type std::pow<int, int>(int, int) [14]
70          0.00    0.00         4/4    inOrderTraversal(node*) [15]
71          0.00    0.00         3/3    OrMatriz(int) [16]
72          0.00    0.00         1/1    sumar() [17]
73          0.00    0.00         1/1    ProductoX(int) [18]
74 -----
75 [2]    63.2    0.24    0.00    20002/20002    main [1]
76          0.24    0.00    20002    Factorial(int) [2]
77 -----
78 [3]    23.7    0.09    0.00    10001/10001    main [1]
79          0.09    0.00    10001    Fibonacci(int) [3]
80 -----
81 [4]    5.3    0.02    0.00         4/4    main [1]
82          0.02    0.00         4    preOrderTraversal(node*) [4]
83 -----
84 [5]    2.6    0.01    0.00    200000/200000    main [1]
85          0.01    0.00    200000    insertNode(node*, int) [5]
86          0.00    0.00    49101/49101    createNode(int) [13]
87 -----
88 [6]    2.6    0.01    0.00         4/4    main [1]
89          0.01    0.00         4    postOrderTraversal(node*) [6]
90 -----
91 [13]    0.0    0.00    0.00    49101/49101    insertNode(node*, int) [5]
92          0.00    0.00    49101    createNode(int) [13]
93 -----
94 [14]    0.0    0.00    0.00    10000/10000    main [1]
95          0.00    0.00    10000    __gnu_cxx::__promote_2<int, int,
96          __gnu_cxx::__promote<int, std::__is_integer<int>::__value>::__type, __gnu_cxx::__promote<int,
97          std::__is_integer<int>::__value>::__type>::__type std::pow<int, int>(int, int) [14]
98 -----
99 [15]    0.0    0.00    0.00         4/4    main [1]
100          0.00    0.00         4    inOrderTraversal(node*) [15]
101 -----
102 [16]    0.0    0.00    0.00         3/3    main [1]
103          0.00    0.00         3    OrMatriz(int) [16]
104 -----
105 [17]    0.0    0.00    0.00         1/1    main [1]
106          0.00    0.00         1    sumar() [17]
107 -----
108 [18]    0.0    0.00    0.00         1/1    main [1]
109          0.00    0.00         1    ProductoX(int) [18]
110 -----
111
112

```

- OPTIMIZADO → [CodigoOptimizado.c](#)

FLAT PROFILE

```

1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4  %   cumulative   self           self       total
5  time   seconds   seconds   calls   ms/call  ms/call  name
6 52.17    0.12    0.12    10001    0.01    0.01  Factorial(int)
7 43.48    0.22    0.10    10001    0.01    0.01  Fibonacci(int)
8  4.35    0.23    0.01         1   10.00   10.00  preorden(node*)
9  0.00    0.23    0.00    50000    0.00    0.00  insertNode(node*, int)
10 0.00    0.23    0.00    31682    0.00    0.00  createNode(int)
11 0.00    0.23    0.00    10000    0.00    0.00  __gnu_cxx::__promote_2<int, int,
    __gnu_cxx::__promote<int, std::__is_integer<int>::__value>::__type, __gnu_cxx::__promote<int,
    std::__is_integer<int>::__value>::__type>::__type std::pow<int, int>(int, int)
12 0.00    0.23    0.00         1    0.00   10.00  arbolBinario()
13 0.00    0.23    0.00         1    0.00    0.00  liberarArbol(node*)
14 0.00    0.23    0.00         1    0.00    0.00  sumaDePunteros()
15 0.00    0.23    0.00         1    0.00    0.00  inOrderTraversal(node*)
16 0.00    0.23    0.00         1    0.00    0.00  arrayMatrizDePunteros()
17 0.00    0.23    0.00         1    0.00    0.00  cubos()
18 0.00    0.23    0.00         1    0.00    0.00  sumar()
19 0.00    0.23    0.00         1    0.00    0.00  matrizX()
20 0.00    0.23    0.00         1    0.00    0.00  OrMatriz(int)
21 0.00    0.23    0.00         1    0.00    0.00  ProductoX(int)
22 0.00    0.23    0.00         1    0.00    0.00  postorden(node*)
23 0.00    0.23    0.00         1    0.00    0.00  quickSort(int*, int, int, int)
24

```


CALL GRAPH (EXPLANATION FOLLOWS)

```

58
59
60          Call graph (explanation follows)
61
62
63 granularity: each sample hit covers 4 byte(s) for 4.35% of 0.23 seconds
64
65 index % time    self  children    called    name
66
67 [1]    100.0    0.00    0.23        10001/10001    main [1]
68          0.12    0.00        10001/10001    Factorial(int) [2]
69          0.10    0.00        10001/10001    Fibonacci(int) [3]
70          0.00    0.01        1/1          arbolBinario() [4]
71          0.00    0.00        1/1          sumar() [20]
72          0.00    0.00        1/1          cubos() [19]
73          0.00    0.00        1/1          ProductoX(int) [23]
74          0.00    0.00        1/1          matrizX() [21]
75          0.00    0.00        1/1          sumaDePunteros() [16]
76          0.00    0.00        1/1          arrayMatrizDePunteros() [18]
77          0.00    0.00        1/1          OrMatriz(int) [22]
78 -----
79          0.12    0.00        10001/10001    main [1]
80 [2]    52.2    0.12    0.00        10001    Factorial(int) [2]
81 -----
82          0.10    0.00        10001/10001    main [1]
83 [3]    43.5    0.10    0.00        10001    Fibonacci(int) [3]
84 -----
85          0.00    0.01        1/1          main [1]
86 [4]    4.3    0.00    0.01        1          arbolBinario() [4]
87          0.01    0.00        1/1          preorden(node*) [5]
88          0.00    0.00        50000/50000    insertNode(node*, int) [12]
89          0.00    0.00        1/1          inOrderTraversal(node*) [17]
90          0.00    0.00        1/1          postorden(node*) [24]
91          0.00    0.00        1/1          liberarArbol(node*) [15]
92 -----
93          0.01    0.00        1/1          preorden(node*) [5]
94          0.01    0.00        1/1          arbolBinario() [4]
95 [5]    4.3    0.01    0.00        1+63364    preorden(node*) [5]
96          63364    preorden(node*) [5]
97 -----
98          0.00    0.00        50000/50000    insertNode(node*, int) [12]
99          0.00    0.00        50000/50000    arbolBinario() [4]
100 [12]    0.0    0.00    0.00        50000+865830    insertNode(node*, int) [12]
101          0.00    0.00        31682/31682    createNode(int) [13]
102          865830    insertNode(node*, int) [12]
103 -----
104          0.00    0.00        31682/31682    insertNode(node*, int) [12]
105 [13]    0.0    0.00    0.00        31682    createNode(int) [13]
106 -----
107          0.00    0.00        10000/10000    cubos() [19]
108 [14]    0.0    0.00    0.00        10000    __gnu_cxx::__promote_2<int, int,
109          __gnu_cxx::__promote<int, std::__is_integer<int>::__value>::__type, __gnu_cxx::__promote<int,
110          std::__is_integer<int>::__value>::__type>::__type std::pow<int, int>(int, int) [14]
111          63364    liberarArbol(node*) [15]
112          0.00    0.00        1/1          arbolBinario() [4]
113 [15]    0.0    0.00    0.00        1+63364    liberarArbol(node*) [15]
114          63364    liberarArbol(node*) [15]
115 -----
116          0.00    0.00        1/1          main [1]
117 [16]    0.0    0.00    0.00        1          sumaDePunteros() [16]

```

En el programa optimizado se puede notar que el tiempo de ejecución es menor, se verifica también que la llamada a la función factorial se redujo en comparación con el programa original.

VALGRIND

Observamos que el programa original tenía un mal manejo del uso de la memoria, habían 3 punteros de los cuales solo uno tenía reservada memoria suficiente para almacenar el entero y asignar su dirección a dicho puntero.

Notamos también que en ninguna parte del código se libera esa memoria asignada.

Se ejecuto el comando **g++ -g CodigoSinOptimizar.c**

Se generó el **a.out** se ejecuta el mismo y por último ejecutamos el valgrind con el siguiente comando **valgrind ./a.out**

- **ORIGINAL** → **CodigoSinOptimizar.c**

```
==21975==
==21975== HEAP SUMMARY:
==21975==   in use at exit: 1,208,428 bytes in 49,105 blocks
==21975==   total heap usage: 49,107 allocs, 2 frees, 1,210,476 bytes allocated
==21975==
==21975== 4 bytes in 1 blocks are definitely lost in loss record 1 of 5
==21975==   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==21975==   by 0x10929A: main (CodigoSinOptimizar.c:33)
==21975==
==21975== 30,000 bytes in 3 blocks are definitely lost in loss record 3 of 5
==21975==   at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==21975==   by 0x109D46: OrMatriz(int) (CodigoSinOptimizar.c:393)
==21975==   by 0x109937: main (CodigoSinOptimizar.c:256)
==21975==
==21975== LEAK SUMMARY:
==21975==   definitely lost: 30,004 bytes in 4 blocks
==21975==   indirectly lost: 0 bytes in 0 blocks
==21975==   possibly lost: 0 bytes in 0 blocks
==21975==   still reachable: 1,178,424 bytes in 49,101 blocks
==21975==     suppressed: 0 bytes in 0 blocks
==21975== Reachable blocks (those to which a pointer was found) are not shown.
==21975== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==21975==
==21975== For lists of detected and suppressed errors, rerun with: -s
==21975== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

En el programa ya optimizado, hicimos que los 3 punteros reserven memoria y que el puntero devuelto por malloc se convierta a puntero de tipo int, y antes de finalizar la función, se libera la memoria usando la función free(), luego de liberar el puntero se establece en null.

- **OPTIMIZADO** → **CodigoOptimizado.c**

```
*****==23460==
==23460== HEAP SUMMARY:
==23460==   in use at exit: 0 bytes in 0 blocks
==23460==   total heap usage: 31,686 allocs, 31,686 frees, 771,400 bytes allocated
==23460==
==23460== All heap blocks were freed -- no leaks are possible
==23460==
==23460== For lists of detected and suppressed errors, rerun with: -s
==23460== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

TIMER

- ORIGINAL → CodigoSinOptimizar.c

Primero se agrega en el fuente

```
#include <time.h>
```

Y dentro del main colocamos:

```
clock_t start= clock();
printf( "Tiempo de procesamiento: %f\n", ( ( double )clock() - start) / CLOCKS_PER_SEC );
```

Ejecutamos el comando `g++ CodigoSinOptimizar.c`
se genera el `a.out` lo ejecutamos y nos calcula el tiempo

```
Tiempo de procesamiento: 0.497141
vboxuser@ubuntu:~/Escritorio/PEF/FINAL PEF 23/Prue
```

- **OPTIMIZADO** → **CodigoOptimizado.c**

Dentro del main colocamos:

```
clock_t start= clock();
printf( "Tiempo de procesamiento: %f\n", ( ( double )clock() - start) / CLOCKS_PER_SEC );
```

Ejecutamos el comando `g++ CodigoSinOptimizar.c`
se genera el `a.out` lo ejecutamos y nos calcula el tiempo

```

999 49999 499992 499900 499900 499998 499992 499992 499990 499994 499990 999992
*****Tiempo de procesamiento: 0.323615
vboxuser@ubuntu:~/Escritorio/PEF/FINAL PEF 23/Pruebas en Ubuntu/OPTIMIZADO/timer
c

```

Conclusión: observamos que el tiempo de ejecución se redujo ya con las optimizaciones en el programa.