



**Ingeniería en Inteligencia Artificial**  
*Algoritmos Bioinspirados*  
**Sem: 2025-1, 5BM1, Práctica 1: Self-Copying Program**  
Fecha: 22 de Septiembre de 2024



## Práctica 1: Self-Copying Program

**Alvarado Sandoval Juan Manuel**  
*jalvarados1900@alumno.ipn.mx*

**Profesor: Juárez Martínez Genaro**

**Resumen:** Este estudio aborda el diseño e implementación de un programa auto-replicante, basado en la teoría de sistemas complejos y evolución artificial propuesta por autores como Melanie Mitchell. El objetivo principal es explorar la capacidad de un programa para copiarse a sí mismo mientras realiza cálculos matemáticos avanzados, como el cubo de una variable  $L$ . Este tipo de sistemas puede desempeñar un papel crucial en el desarrollo de algoritmos bioinspirados que simulan comportamientos adaptativos y evolutivos. A lo largo del documento se presentan los resultados obtenidos, destacando la robustez y eficiencia del programa al mantener su integridad funcional en múltiples ciclos de replicación, evidenciando su potencial para aplicaciones en inteligencia artificial y computación evolutiva.

**Palabras Clave:** *Auto-replicación de código, computación evolutiva, algoritmos bioinspirados, sistemas complejos, adaptación digital, inteligencia artificial evolutiva.*

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Planteamiento del Problema</b>	<b>3</b>
<b>3. Conceptos Básicos</b>	<b>4</b>
3.1. Auto-replicación . . . . .	4
3.2. Algoritmos Bioinspirados . . . . .	4
3.3. Computación Evolutiva . . . . .	4
3.4. Sistemas Complejos . . . . .	4
3.5. Adaptación Digital . . . . .	5
3.6. Vida Artificial . . . . .	5
3.7. Adaptación en Sistemas Complejos . . . . .	5
<b>4. Pseudo-códigos</b>	<b>6</b>
4.1. Pseudo-código de "self-copying program" de Mitchell M. . . . .	6
4.2. Pseudo-código del problema propuesto en esta práctica . . . . .	6
<b>5. Desarrollo</b>	<b>6</b>
5.1. Experimentación y Resultados . . . . .	6
5.2. Código . . . . .	8
<b>6. Conclusión</b>	<b>10</b>
<b>7. Bibliografía</b>	<b>11</b>

# 1 Introducción

La auto-replicación en la computación ha sido un tema de interés desde que John von Neumann propuso su concepto de autómatas auto-replicantes. Este desafío implica no solo la capacidad de un sistema para generar copias exactas de sí mismo, sino también para incorporar variaciones que permitan la evolución y adaptación del sistema en entornos cambiantes. Melanie Mitchell, en su análisis de la evolución y la vida en las computadoras, señala que la reproducción y adaptación de sistemas computacionales no es solo teóricamente posible, sino que tiene profundas implicaciones para nuestra comprensión de la inteligencia y la vida artificial.

En la evolución computacional, los algoritmos genéticos, como los propuestos por John Holland, proporcionan un marco para explorar cómo las estrategias de búsqueda pueden evolucionar y adaptarse a través de generaciones de individuos digitales. Estos algoritmos utilizan principios de selección natural para mejorar iterativamente las soluciones a problemas específicos, lo que refleja cómo las poblaciones biológicas se adaptan a su entorno. Esta capacidad de auto-mejora y adaptación es esencial para desarrollar sistemas autónomos en inteligencia artificial que puedan operar de manera efectiva en condiciones inciertas.

En este contexto, se propone un programa que no solo se auto-replica, sino que también realiza operaciones matemáticas, como el cálculo del cubo de una variable  $L$ . Este enfoque permite estudiar cómo los principios de auto-replicación y ejecución de tareas computacionales pueden integrarse en un sistema cohesivo.

El marco teórico de este estudio se apoya en pilares principales: los autómatas auto-replicantes de von Neumann y los sistemas complejos descritos por Melanie Mitchell.

Los autómatas auto-replicantes de von Neumann, por otro lado, proporcionan una estructura conceptual para entender cómo un sistema puede generar copias exactas de sí mismo. Von Neumann mostró que, para evitar la regresión infinita en la replicación, es necesario que el sistema contenga una descripción completa de sí mismo y de las instrucciones para replicarse. Este concepto se ha expandido en la computación moderna para incluir mecanismos de auto-reproducción y adaptación en sistemas más complejos.

Finalmente, los sistemas complejos, tal como los describe Melanie Mitchell, exploran cómo interacciones simples entre componentes pueden dar lugar a comportamientos emergentes complejos.

## 2 Planteamiento del Problema

El problema consiste en diseñar un programa que, además de auto-replicarse, realice el cálculo del cubo de una variable  $L$ . Este desafío implica combinar la gestión de memoria y recursos para la copia fiel del programa, con la ejecución de operaciones matemáticas en cada ciclo de replicación. Se deben prever errores que puedan comprometer la integridad del código, así como conflictos en la gestión de recursos cuando se ejecutan múltiples instancias del programa.

El enfoque del problema también debe considerar cómo implementar mecanismos de adaptación que permitan al programa ajustarse a variaciones en su entorno computacional. Esto incluye la capacidad de manejar cambios en la estructura del código y en la configuración del entorno de ejecución sin perder la funcionalidad básica de replicación y cálculo. La capacidad de adaptación es un componente esencial en la evolución digital, ya que permite que los sistemas se ajusten a nuevas condiciones sin intervención externa.

## 3 Conceptos Básicos

### 3.1 *Auto-replicación*

La auto-replicación es la capacidad de un programa o sistema de generar una copia exacta de sí mismo, manteniendo todas sus características y funcionalidades. Este concepto, inspirado en procesos biológicos como la replicación del ADN, plantea desafíos computacionales significativos, como evitar regresiones infinitas y garantizar que cada copia conserve la integridad del código original. La auto-replicación ha sido estudiada ampliamente en el contexto de los autómatas celulares y la vida artificial, destacando las contribuciones de John von Neumann en la teoría de sistemas auto-replicantes.

### 3.2 *Algoritmos Bioinspirados*

Los algoritmos bioinspirados son técnicas computacionales que imitan procesos naturales para resolver problemas complejos. Entre los más comunes se encuentran los algoritmos genéticos, las redes neuronales artificiales y los sistemas inmunológicos artificiales. Estos algoritmos buscan emular la capacidad de adaptación, aprendizaje y evolución de los sistemas biológicos, aplicándolos a contextos computacionales para desarrollar soluciones innovadoras y eficientes.

### 3.3 *Computación Evolutiva*

La computación evolutiva es un subcampo de la inteligencia artificial que utiliza principios de la evolución biológica, como la selección, la reproducción y la mutación, para desarrollar algoritmos que buscan soluciones óptimas a problemas específicos. A través de la generación de poblaciones de

posibles soluciones y la aplicación de operadores genéticos, estos algoritmos permiten la evolución de soluciones que se adaptan mejor a la función objetivo definida, optimizando el rendimiento del sistema con cada iteración.

### ***3.4 Sistemas Complejos***

Los sistemas complejos se caracterizan por la interacción de numerosos componentes interdependientes que, a través de sus interacciones, generan comportamientos emergentes y dinámicas globales que no se pueden predecir directamente a partir de las propiedades individuales de los componentes. Ejemplos de sistemas complejos incluyen ecosistemas, mercados financieros y redes neuronales. En computación, el estudio de estos sistemas ayuda a entender cómo interacciones locales simples pueden producir patrones globales complejos y organizados.

### ***3.5 Adaptación Digital***

La adaptación digital se refiere a la capacidad de un sistema computacional para modificar su comportamiento en respuesta a cambios en su entorno o en sus propios objetivos. Esta capacidad de adaptación es fundamental en la evolución digital y en los algoritmos bioinspirados, donde se busca que los programas o agentes sean capaces de aprender y mejorar su desempeño sin intervención humana directa, optimizando su estructura y comportamiento para satisfacer nuevas condiciones o requisitos.

### ***3.6 Vida Artificial***

La vida artificial es un campo interdisciplinario que estudia los principios fundamentales de la vida a través de la simulación y la creación de sistemas artificiales. En computación, la vida artificial se centra en el diseño de algoritmos y modelos que replican aspectos de los seres vivos, como la reproducción, la evolución y el comportamiento adaptativo. Esto permite explorar preguntas fundamentales sobre la naturaleza de la vida y la inteligencia, así como desarrollar nuevas tecnologías basadas en principios biológicos.

### ***3.7 Adaptación en Sistemas Complejos***

La adaptación en sistemas complejos se refiere a la capacidad de un sistema para modificar su estructura o comportamiento en respuesta a perturbaciones externas o internas, garantizando su supervivencia y eficiencia operativa. En el contexto de la computación, esto implica que un sistema puede ajustarse de manera autónoma a variaciones en su entorno, optimizando sus recursos y estrategias para cumplir con sus objetivos.

## 4 Pseudo-códigos

### 4.1 Pseudo-código de "self-copying program" de Mitchell M.

---

**Algorithm 1** Self-Copying Program

---

```
1:  $L \leftarrow mem + 1$ 
2: print("program copy")
3: print("L = mem + 1;")
4: while line[L]  $\neq$  end do
5:   print(line[L])
6:    $L \leftarrow L + 1$ 
7: end while
8: print(end)
```

---

### 4.2 Pseudo-código del problema propuesto en esta práctica

---

**Algorithm 2** Self-Copying Program-Cube

---

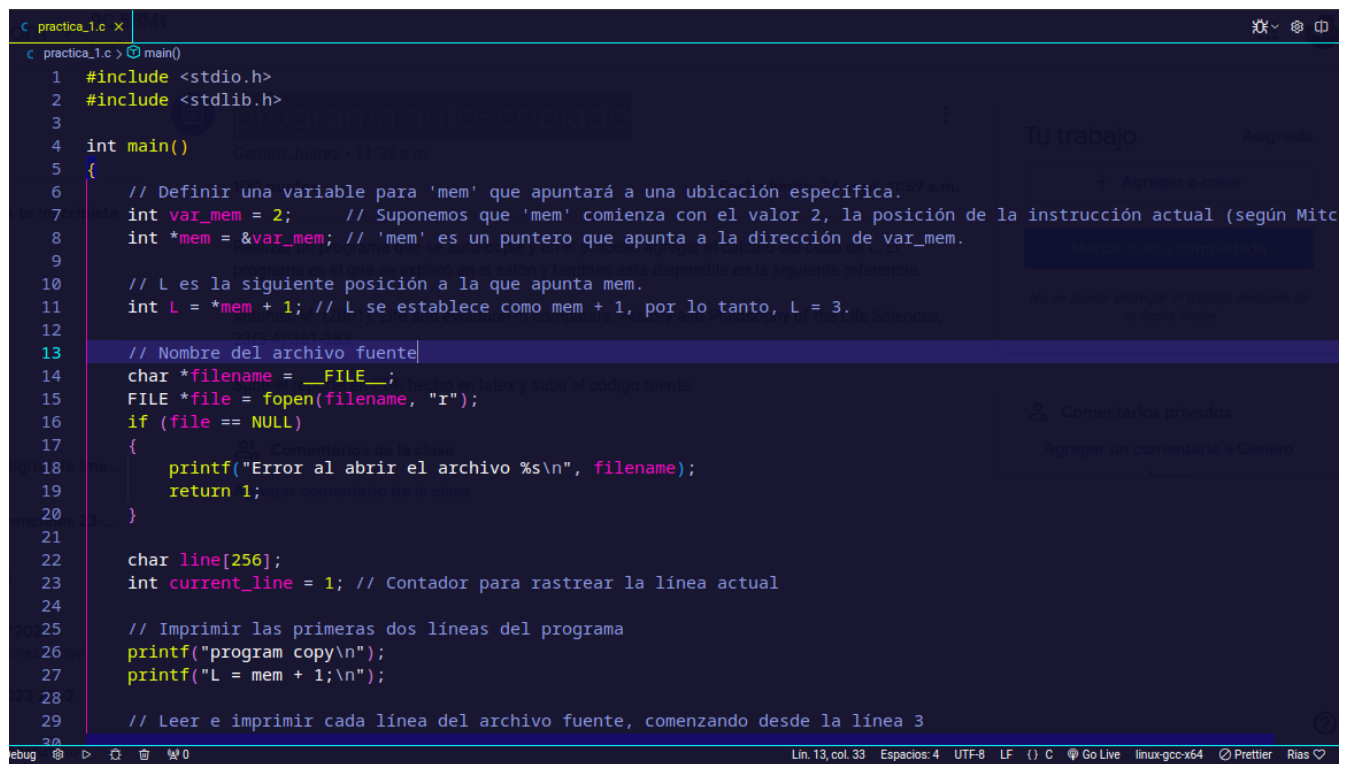
```
1: ALGORITHM Self-Copying-Program-Cube
2:  $L \leftarrow mem + 1$ 
3: Print("program copy")
4: Print("L = mem + 1;")
5: while line[L]  $\neq$  end do
6:   Print(line[L])
7:    $Cube \leftarrow L^3$ 
8:   Print(Cubo de L (" , L, ") es: " , Cube)
9:    $L \leftarrow L + 1$ 
10: end while
11: Print(end)
12: END ALGORITHM
```

---

## 5 Desarrollo

### 5.1 Experimentación y Resultados

En esta sección se muestra el proceso de auto-replicación de un programa que, además de copiarse a sí mismo, calcula el cubo de una variable L en cada iteración. A continuación se presentan los resultados de diferentes configuraciones del programa, evaluando su rendimiento y eficiencia en la replicación.

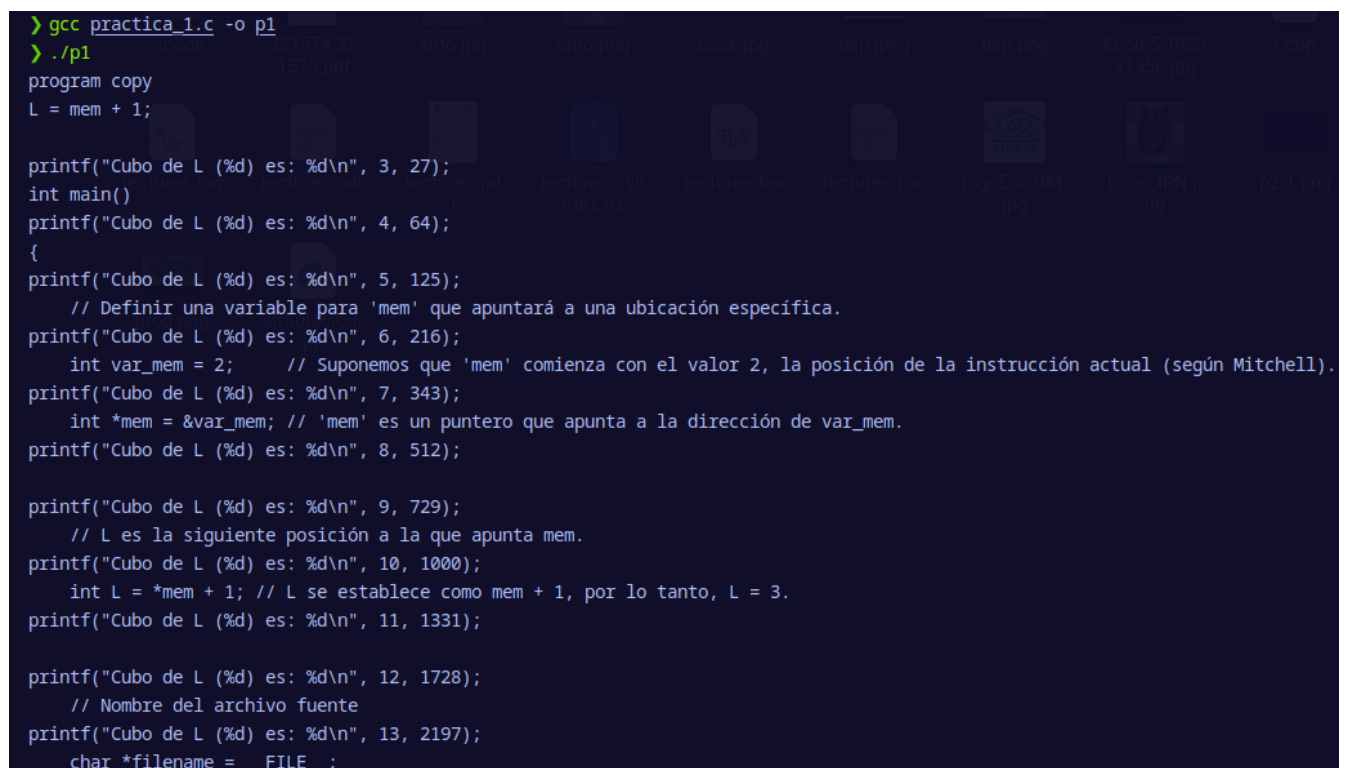


```

c practica_1.c x
c practica_1.c > main()
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      // Definir una variable para 'mem' que apuntará a una ubicación específica.
7      int var_mem = 2; // Suponemos que 'mem' comienza con el valor 2, la posición de la instrucción actual (según Mitc
8      int *mem = &var_mem; // 'mem' es un puntero que apunta a la dirección de var_mem.
9
10     // L es la siguiente posición a la que apunta mem.
11     int L = *mem + 1; // L se establece como mem + 1, por lo tanto, L = 3.
12
13     // Nombre del archivo fuente
14     char *filename = __FILE__;
15     FILE *file = fopen(filename, "r");
16     if (file == NULL)
17     {
18         printf("Error al abrir el archivo %s\n", filename);
19         return 1;
20     }
21
22     char line[256];
23     int current_line = 1; // Contador para rastrear la línea actual
24
25     // Imprimir las primeras dos líneas del programa
26     printf("program copy\n");
27     printf("L = mem + 1;\n");
28
29     // Leer e imprimir cada línea del archivo fuente, comenzando desde la línea 3

```

Figura 1: Script original comentado del programa.



```

> gcc practica_1.c -o p1
> ./p1
program copy
L = mem + 1;

printf("Cubo de L (%d) es: %d\n", 3, 27);
int main()
printf("Cubo de L (%d) es: %d\n", 4, 64);
{
printf("Cubo de L (%d) es: %d\n", 5, 125);
    // Definir una variable para 'mem' que apuntará a una ubicación específica.
printf("Cubo de L (%d) es: %d\n", 6, 216);
    int var_mem = 2; // Suponemos que 'mem' comienza con el valor 2, la posición de la instrucción actual (según Mitchell).
printf("Cubo de L (%d) es: %d\n", 7, 343);
    int *mem = &var_mem; // 'mem' es un puntero que apunta a la dirección de var_mem.
printf("Cubo de L (%d) es: %d\n", 8, 512);

printf("Cubo de L (%d) es: %d\n", 9, 729);
    // L es la siguiente posición a la que apunta mem.
printf("Cubo de L (%d) es: %d\n", 10, 1000);
    int L = *mem + 1; // L se establece como mem + 1, por lo tanto, L = 3.
printf("Cubo de L (%d) es: %d\n", 11, 1331);

printf("Cubo de L (%d) es: %d\n", 12, 1728);
    // Nombre del archivo fuente
printf("Cubo de L (%d) es: %d\n", 13, 2197);
    char *filename = __FILE__;

```

Figura 2: Ejecución del programa. Como salida se observa que el programa replica sus líneas de código y según el estado de L, calcula el cubo del mismo.

```

> gcc practica_1.c -o p1
> ./p1
program copy
L = mem + 1;

printf("Cubo de L (%d) es: %d\n", 3, 27);
int main()
printf("Cubo de L (%d) es: %d\n", 4, 64);
{
printf("Cubo de L (%d) es: %d\n", 5, 125);
    // Definir una variable para 'mem' que apuntará a una ubicación específica.
printf("Cubo de L (%d) es: %d\n", 6, 216);
    int var_mem = 2;    // Suponemos que 'mem' comienza con el valor 2, la posición de la instrucción actual (según Mitchell).
printf("Cubo de L (%d) es: %d\n", 7, 343);
    int *mem = &var_mem; // 'mem' es un puntero que apunta a la dirección de var_mem.
printf("Cubo de L (%d) es: %d\n", 8, 512);

printf("Cubo de L (%d) es: %d\n", 9, 729);
    // L es la siguiente posición a la que apunta mem.
printf("Cubo de L (%d) es: %d\n", 10, 1000);
    int L = *mem + 1; // L se establece como mem + 1, por lo tanto, L = 3.
printf("Cubo de L (%d) es: %d\n", 11, 1331);

printf("Cubo de L (%d) es: %d\n", 12, 1728);
    // Nombre del archivo fuente
printf("Cubo de L (%d) es: %d\n", 13, 2197);
    char *filename = __FILE__;

```

Figura 3: Verificación de las siguientes líneas de código y la impresión del \*end\*.

Como se observa, las figuras proporcionan una visión clara del funcionamiento del programa en cada etapa. En la primera figura, se observa la estructura lógica y los comentarios que explican cada paso del programa. La segunda figura evidencia la ejecución dinámica del programa, replicando y realizando cálculos simultáneamente. Finalmente, la tercera figura verifica que el programa ha concluido con éxito su operación, sin errores en la copia y ejecución de instrucciones.

## 5.2 Código

Se describe el código fuente del programa, con un enfoque en cómo se realiza la auto-replicación y el cálculo del cubo de  $L$ , manteniendo la estructura y funcionalidad del código en cada instancia generada.

```

1      #include <stdio.h>
2      #include <stdlib.h>
3
4      int main()
5      {
6          // Definir una variable para 'mem' que apuntara a una ubicacion
          especifica.
7          int var_mem = 2;    // Suponemos que 'mem' comienza con el
          valor 2, la posicion de la instruccion actual (segun Mitchell
          ).

```



```
8      int *mem = &var_mem; // 'mem' es un puntero que apunta a la
          direccion de var_mem.
9
10     // L es la siguiente posicion a la que apunta mem.
11     int L = *mem + 1; // L se establece como mem + 1, por lo tanto,
          L = 3.
12
13     // Nombre del archivo fuente
14     char *filename = __FILE__;
15     FILE *file = fopen(filename, "r");
16     if (file == NULL)
17     {
18         printf("Error al abrir el archivo %s\n", filename);
19         return 1;
20     }
21
22     char line[256];
23     int current_line = 1; // Contador para rastrear la linea actual
24
25     // Imprimir las primeras dos lineas del programa
26     printf("program copy\n");
27     printf("L = mem + 1;\n");
28
29     // Leer e imprimir cada linea del archivo fuente, comenzando
          desde la linea 3
30     while (fgets(line, sizeof(line), file))
31     {
32         // Empezar a copiar desde la linea 3 en adelante
33         if (current_line >= 3)
34         {
35             // Imprimir la linea actual del codigo fuente
36             printf("%s", line);
37
38             // Calcular e imprimir el cubo de L para cada linea
          copiada
39             int cube = L * L * L; // Calcular el cubo de L
40             printf("printf(\"Cubo de L (%d) es: %d\\n\", %d, %d);\n",
          L, cube);
41             L++;
42         }
43         current_line++;
44     }
45
46     // Imprimir "end" para finalizar la copia
47     printf("end\n");
48
49     fclose(file);
50     return 0;
51 }
```

---

Listing 1: Programa auto-replicado con cálculo de cubo

## 6 Conclusión

La auto-replicación en la computación no solo es posible, sino que abre nuevas vías para entender y aplicar conceptos evolutivos en sistemas artificiales. Este estudio demuestra que es factible crear programas que se adapten y mantengan su funcionalidad, inspirando futuros desarrollos en inteligencia artificial y vida artificial.

Como estudiante, haber logrado implementar un programa auto-replicante con capacidad de cálculo adicional ha sido una experiencia enriquecedora y motivadora. Me ha permitido comprender un poco de cómo las ideas teóricas de auto-replicación y adaptación pueden transformarse en soluciones prácticas y funcionales.

## 7 Bibliografía

1. Mitchell, M. (2001). *Life and evolution in computers*. History and Philosophy of the Life Sciences.
2. Holland, J.H. (1992). *Adaptation in Natural and Artificial Systems*.
3. Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*.