Lab. 1, Programación, algoritmia y rendimiento

Martínez Hernández, Dafne. angelica@ciencias.unam.mx
Olvera Trejo, Alberto. alberto0410@ciencias.unam.mx

30 de junio de 2022

Resumen: En el presente trabajo se muestran dos programas utilizando clases y funciones *lambda*; uno de ellos en un conversor de unidades y el otro un simulador de la tercera Guerra Mundial. En el segundo apartado se realizan análisis de algoritmos que emplean las estrategias de recesión y dividir y vender, como lo son las torres de Hanoi y el problema de los triominós, respectivamente.

Palabras clave: programación, algoritmos, recursión.

1. Introducción

La programación orientada a objetos (POO) se basa en abstraer el problema y crear un modelo para el problema a resolver, de esta manera se puede reutilizar el código creado y disminuye el número de errores.

Por otro lado, las funciones lambda son producto del trabajo del cálculo lambda de Alonzo Church, se originaron antes de las computadoras. Son usadas cuando una función únicamente se ocupa una o muy pocas veces; son más sencillas que una función normal

Un algoritmo es una detallada serie de instrucciones concretas con las cuales se encuentra la solución a un problema dado. Los algoritmos son el objeto de estudio de la algoritmia, por lo que esta última hace referencia a la búsqueda de soluciones a un problema concreto.

Las Torres de Hanoi es juego matemático inventado en 1883 por el matemático francés Édouard Lucas. El juego consiste en un número n de discos perforados de radio creciente que se apilan insertándose en uno de los tres postes fijados a un tablero. El objetivo del juego es trasladar la torre de discos a otro de los postes, teniendo en cuenta que sólo se puede mover un disco a la vez y que no se puede poner un disco más grande encima de uno más pequeño.

Un poliominó es una figura geométrica plana que se forma conectando dos o más cuadrados por alguno de sus lados. Los cuadrados se conectan lado con lado, sin embargo, éstos no se pueden conectar por sus vértices o juntando sólo parte de un lado de un cuadrado con parte de un lado de otro. [1] Dicho esto, un triominó es una figura geométrica de 3 cuadrados, uno de los tipos de triominós que se forman son los L-Triominó, y son aquellos que, como su nombre lo dice, forman una ”L”, por lo que una pieza de L-Triominó se compone de un cuadrado de 2×2 con una pieza perdida.

2. Metodología

2.1. Conversor de unidades

En el conversor de unidades se creó una clase para cada tipo de unidad (tiempo, distancia y temperatura), y cada una de ellas contiene funciones lambda para poder convertir de una unidad a otra y hacer su inversa. Posteriormente se creó un método cuyo único parámetro es un índice; dentro se crea un diccionario que contiene como claves las funciones ya mencionadas y como valor las unidades que retornan. Finalmente se crea un método de clase que va a servir como menú en donde se le pregunta al usuario las unidades y la magnitud que ingresa, después se le muestran las posibles conversiones que hay y la opción de salida Una vez ingresada la respuesta, el programa va al diccionario y llama a la función para poder regresar como resultado el nombre de la unidad a la que se convirtió y la magnitud.

2.2. Tercera Guerra Mundial

Para el ejercicio de la Tercera Guerra Mundial se necesitaron los módulos de Python *numpy, matplotlip.pyplot y random*; se crearon 3 clases, las cuales hacen referencia a los submarinos, el cañón, las balas del cañón , respectivamente. Para cada clase se declararon distintos métodos: **Submarinos,** el método *mover* permitía que los submarinos avanzaran, mientras que el método *estar_dentro* permite evaluar cuando el submarino pase por el radar; **Cañón:**, el método *graf_ radar* regresa 2 listas que servirán para poder graficar un semicírculo con centro en el cañón.

Una vez definidas las clases y los métodos necesarios, se empezó a hacer la animación, la cual consiste en una serie de imágenes consecutivas hechas con el comando *plt.pause()*. Cada vez que se grafica una nueva imagen, los submarinos activos caminan 0.08 unidades hacia abajo. En la parte inferior aparece un semicirulo, el cual representa el radar, y cuando un submarino entra ahí, el cañón verifica si su tiempo de enfriamiento ha pasado y en caso de ser positivo, entonces elimina al objetivo y se imprime en pantalla "Un submarino ha sido eliminado". Cuando la coordenada y de un barco es menor que cero, entonces el programa imprime en pantalla "Un barco ha entrado". La animación termina cuando se ha superado un número de 200 imágenes

2.3. Torres de Hanoi

Al analizar el problema de las Torres de Hanoi se puede notar que se trata de recursión, pues hay un caso base, el cual es mover el primer disco de la torre inicial a la final, y todos los demás se derivan de él. Cuando tenemos un número n de discos, el algoritmo es pasar los $n-1$ discos de la torre inicial a la auxiliar, la cual es la de en medio (es decir, hacer el algoritmo para $n-1$ donde los papeles de torre auxiliar y torre final se intercambian), posteriormente se pasa el disco más grande a la torre final y finalmente se realiza nuevamente el algoritmo para pasar los $n-1$ discos de la torre auxiliar a la torre final.
Dicho lo anterior, es claro que el numero de movimientos que se requieren para resolver un juego de n discos ($T(n)$) son:

$$T(n) = \begin{cases} 1 & si \quad n = 1 \\ 2T(n-1) + T(1) & si \quad n \neq 1 \end{cases}$$

2.3.1. Análisis del algoritmo

Sea $T(n)$ el número de movimientos que se necesitan para resolver un juego con n discos.

Caso base:
$T(1) = 1$

$$\begin{aligned} T(n) &= 2T(n-1) + T(1) \quad def.\ de\ T(n) \\ &= 2T(n-1) + 1 \quad def.\ de\ T(1) \\ &= 2(2T(n-2)+1) + 1 \quad def.\ de\ T(n-1) \\ &= 2^2T(n-2) + 2 + 1 \quad distributividad \\ &= 2^2(2T(n-3)+1) + 2 + 1 \quad def.\ de\ T(n-2) \\ &= 2^3T(n-3) + 2^2 + 2 + 1 \quad distributividad \end{aligned}$$

$k\ veces$:

$$= 2^kT(n-k) + 2^{k-1} + \cdots + 2 + 1$$

$n-1\ veces$:

$$\begin{aligned} &= 2^{n-1}T(n-(n-1)) + 2^{n-1-1} + \cdots + 2 + 1 \\ &= 2^{n-1}T(1) + 2^{n-2} + \cdots + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\ &= \sum_{i=0}^{n-1} 2^i \\ &= \left(\frac{1-2^{(n-1)+1}}{1-2}\right) suma\ term.\ en\ serie\ geom. \\ &= \left(\frac{1-2^n}{-1}\right) \\ &= 2^n - 1 \end{aligned}$$

Con esto se concluye que el número de pasos que se necesitan para resolver un juego de n discos es $T(n) = 2^n - 1$.

2.3.2. Orden de complejidad de algoritmo

Para acotar la función $T(n) = 2^n - 1$, se tiene en cuenta que $-1 \leq 0$ y que $\forall n \in \mathbb{N}\ \ 2^n \leq 2^n$ por lo que, usando la transitividad del orden, se llega a que $\forall n \in \mathbb{N}\ \ 2^n - 1 \leq 2^n$.

Considerando $c = 1$ y $g(x) = 2^n$, por la definición de Orden de complejidad computacional, es claro que $T(n) \in O(2^n)$ con respecto al número de movimientos

2.3.3. Manejo de memoria

Tomando en cuenta el programa hecho para este problema, se tiene que al tomar $discos = 963$, Python marca *maximum recursion depth exceeded while calling a Python object*, por lo tanto se concluye que el límite de recursión es 963
Por lo argumentado arriba, el número de operaciones que hace el programa dándole un número de entrada n está dado por:

$$T(n) = 2^n - 1$$

2.4. Triominós

Finalmente, para el problema de los triominós su usaron las estrategias de dividir y vencer y recursión. Únicamente se trabajará el caso de los tableros de $2^n \cdot 2^n$ ya que con esta disposición se puede generar un método para resolver el problema.
Se demostrará brevemente que todo tablero deficiente (es decir, que le falta un cuadrado) de $2^n \cdot 2^n$ se puede embaldosar con piezas de triominós. Claramente para $n = 1$ se puede lograr, pues únicamente se tiene que colocar una pieza. Luego, supongamos que se vale para cualquier número $k \leq n$ y veamos que se puede para n. Consideremos un tablero deficiente de $2^n \cdot 2^n$ y se procede a dividirlo en 4 tableros iguales, es decir que cada uno va a medir $2^{n-1} \cdot 2^{n-1}$. Después, se selecciona el tablero en el cuál se encuentra la pieza faltante, como $n-1 < n$, entonces se puede llenar ese tablero y ahora únicamente nos faltan 3. Sin pérdida de la generalidad, supongamos que el tablero mencionado estaba en la parte superior derecha, como se muestra en la imagen 1 Luego, se coloca una pieza de triominó en el centro, justo como se muestra en al figura 1. De esta manera se obtienen 3 tableros deficientes de $2^{n-1} \cdot 2^{n-1}$ y como estos ya se pueden resolver, entonces es posible embaldosar el de $2^n \cdot 2^n$.

Por lo tanto, todo tablero de $2^n \cdot 2^n$ puede ser embaldosado con piezas de triominó.

Ahora se calculará la función $T(n)$, es decir, el

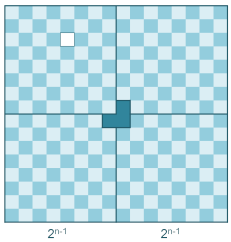

Figura 1: Tablero de $2^n \cdot 2^n$ dividido en cuatro tableros [1]

número de operaciones para un tablero deficiente de $2^n \cdot 2^n$.

Si $n = 1$, entonces se realiza un único movimientos. Si $n > 1$ entonces se divide el tablero en 4, colocamos una pieza en medio y finalmente se resuelve las 4 partes. De esta manera:

$$T(n) = \begin{cases} 1 & si \quad n = 1 \\ 4T(n-1) + T(1) & si \quad n \neq 1 \end{cases}$$

2.4.1. Análisis del algoritmo

Sea $T(n)$ el número de movimientos que se necesitan para resolver un juego con n discos.

Caso base:
$T(1) = 1$

$$\begin{aligned} T(n) &= 4T(n-1) + T(1) && def.\ de\ T(n) \\ &= 4T(n-1) + 1 && def.\ de\ T(1) \\ &= 4(4T(n-2) + 1) + 1 && def.\ de\ T(n-1) \\ &= 4^2T(n-2) + 4 + 1 && distributividad \\ &= 4^2(4T(n-3) + 1) + 4 + 1 && def.\ de\ T(n-2) \\ &= 4^3T(n-3) + 4^2 + 4 + 4^0 && distributividad \end{aligned}$$

k veces

$$= 4^kT(n-k) + 4^{k-1} + ... + 4^2 + 4 + 4^0$$

$n+1$ *veces*

$$
\begin{aligned}
&= 4^{n-1}T(n-(n-1)) + 4^{(n-1)-1} + ... + 4^0 \\
&= 4^{n-1}T(1) + 4^{(n-1)-1} + ... + 4 + 4^0 \\
&= \sum_{i=0}^{n-1} 4^i \\
&= \left(\frac{1-4^{(n-1)+1}}{1-4}\right) \textit{suma serie geom.} \\
&= \left(\frac{1-4^n}{-3}\right) \\
&= \frac{4^n}{3} - \frac{1}{3}
\end{aligned}
$$

Con esto se concluye que el número de movimientos que se necesitan para resolver un juego con tablero de $2^n \cdot 2^n$ son $T(n) = \frac{4^n}{3} - \frac{1}{3}$.

2.4.2. Orden de complejidad de algoritmo

Para acotar la función $T(n) = \frac{4^n}{3} - \frac{1}{3}$;

- $-\frac{1}{3} \leq 0$
- $\frac{1}{3} \leq 1$
- $\forall n \in \mathbb{N} \;\; 4^n \leq 4^n$

Utilizando las propiedades de las desigualdades, Por el punto 2 y 3 se tiene que

$$\forall n \in \mathbb{N} \;\; \frac{4^n}{3} \leq 4^n$$

Y usando el punto uno llegamos a que

$$\forall n \in \mathbb{N} \;\; \frac{4^n}{3} - \frac{1}{3} \leq 4^n$$

$$\therefore \quad T(n) \leq 4^n$$

Si tomamos $c = 1$ y $g(x) = 4^n$, por la definición de Orden de complejidad computacional, es claro que $T(n) \in O(4^n)$ *(orden polinomial)* con respecto al número de movimientos.

2.4.3. Manejo de memoria

Tomando en el cuenta el programa hecho, se tiene que el límite de recursividad es cuando $n = 963$, a partir de ese momento Python marca *maximum recursion depth exceeded while calling a Python object*

Puesto que el programa creado está hecho de manera recursiva, entonces el manejo de memoria tiene un orden exponencial

3. Resultados y discusión

3.1. Conversor de unidades

El menú creado para las unidades de tiempo es el que se muestra en la siguiente imagen 2.

Como se puede ver, cada conversión incluye su inversa, *i.e* existe la posibilidad de pasar de años a semanas y de semanas a años. También incluye la opción de poder salir del menú.
El resultado que arroja el programa es la equivalencia de las unidades de entrada a la solicitada.

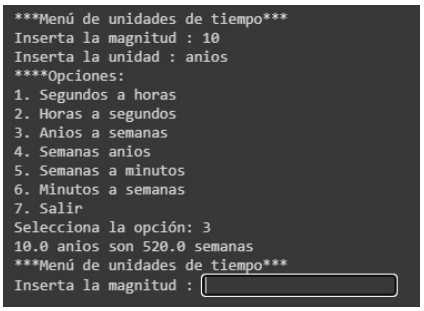

Figura 2: Menú de unidades de tiempo

3.2. Torres de Hanoi

Se concluye que el algoritmo tiene una complejidad de orden exponencial tanto en tiempo de resolución como en manejo de memora, por lo que si bien es "fácil" de resolver el problema, se vuelve repetitivo y muy tardado de hacer, además de que gasta una cantidad exagerada de memoria del ordenador, por lo que no es buena idea meter un número muy grande

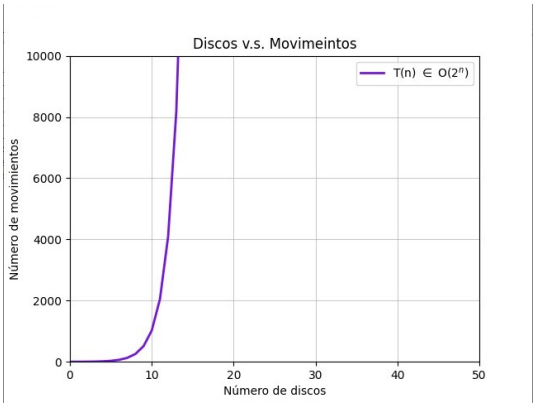

Figura 3: Orden de complejidad

3.3. Tercera Guerra Mundial

Una parte relevante del programa es mostrar el momento exacto en el que se elimina un submarino, tal como se muestra en la imagen 4

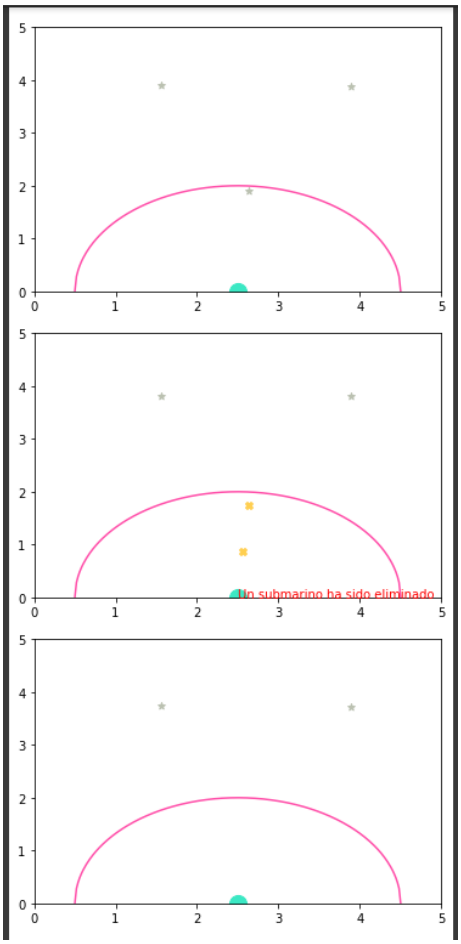

Figura 4: Subamrino eliminado

La secuencia de imágenes muestra cuando un submarino entra al radar, después cuando el cañon le dispara y finalmente cuando el submarino desaparece del rada

3.4. Triominós

Análogamente que al problema anterior, se concluye que al aumentar el número ingresando al programa, éste se vuelve mucho más tardado de resolver, puede tardar muchos minutos en terminar de ejecutarse el código.

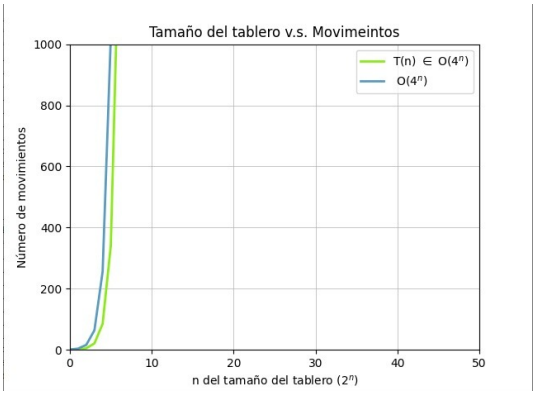

Figura 5: Orden de complejidad

4. Conclusión

El uso de clases y funciones *lambda* en los primeros dos ejercicios ayudaron a tener un mejor control sobre el código y a evitar errores, puesto que se tenía bien definido qué cosa iba a hacer cada objeto y si había un error era más rápido de identificar.
En cuanto a los dos últimos ejercicios, se tiene que si bien el método recursivo es relativamente fácil de implementar, en la practica no es la mejor opción cuando se tiene un número considerable de datos. Por otro lado, la estrategia *dividir y vencer* fue de gran ayuda al plantear y pensar cómo resolver el problema de los triominós, ya que basta en concentrarse en cómo resolver un problema chico para así poder resolver un problema más grande.

Referencias:

[1] Cuaderno de Cultura Científica. Jul. de 2014. URL: https://culturacientifica.com/2014/07/16/embaldosando-con-l-triominos-un-ejemplo-de-demostracion-por-induccion/.