Ejercicios iniciales (Solución)

Creado por: Grupo 04

1. **Ejercicio 4**

   (a) Abstracción para obtener propiedades genéricas; por encima de los detalles.

   Solución.
   La abstracción es el proceso de generalizar al reducir el contenido de información de un concepto o un fenómeno observable. Consiste en captar las características esenciales de un objeto, así como su comportamiento, ignorando otras propiedades del objeto en cuestión. La abstracción es una habilidad esencial para la construcción de modelos y la descomposición de problemas.

   La abstracción sirve para la formación del conocimiento humano, ya que permite crear conceptos abstractos a partir de la observación de la realidad. También sirve para aspectos informáticos, como la programación orientada a objetos, ya que permite separar el comportamiento específico de un objeto y enfocarse en su visión externa. La abstracción facilita el análisis y la comprensión de los fenómenos, sistemas o procesos.

   Algunos ejemplos de abstracción son:

   1. Representar objetos cotidianos con palabras o imágenes.
   2. Elaborar definiciones de conceptos como la democracia o la justicia.
   3. Diagnosticar una enfermedad a partir de los síntomas de un paciente.
   4. Identificar el problema de un vehículo a partir de los sonidos o las señales que emite.
   5. Clasificar animales según sus características comunes.

   (b) Lenguaje WHILE

   Solución.
   El lenguaje WHILE es un lenguaje de programación simple construido a partir de asignaciones , composición secuencial, condicionales y sentencias while. Es un lenguaje imperativo simple con variables enteras y expresiones enteras y booleanas. Para ramificación tiene una declaración if-else y para bucle tiene una declaración while.

   El lenguaje WHILE es Turing computable, lo que significa que puede resolver cualquier problema que pueda ser resuelto por una máquina de Turing. En teoría, esto significa que el lenguaje WHILE puede resolver cualquier problema computable. Sin embargo, en la práctica, algunos problemas pueden ser muy difíciles de resolver con el lenguaje WHILE debido a limitaciones de tiempo y espacio.

   La definición formal del lenguaje WHILE es la siguiente:

Dados los alfabetos:
$\Sigma_w = \{\texttt{X}, :=, 1, +, -, \texttt{while}\ , \neq, 0, \texttt{do}\ , \texttt{od}, ;\}$
$\Sigma_d = \{0, 1, 2, 3, 4; 5, 6, 7, 8, 9\}$

Sea $G = (N, T, P, <\text{code}>)$ una gramática con:

$N = \{<\text{number}>, <\text{firstdigit}>, <\text{restnumber}>, <\text{digit}>,$
$<\text{code}>, <\text{sentence}>, <\text{assignment}>, <\text{loop}>, <\text{id}>\}$
$T = \Sigma_d \cup \Sigma_w$
$P = \{$

$$
\begin{array}{lcl}
<\text{number}> & \rightarrow & <\text{firstdigit}> \mid <\text{firstdigit}><\text{restnumber}>, \\
<\text{firstdigit}> & \rightarrow & 1|2|3|4|5|6|7|8|9, \\
<\text{digit}> & \rightarrow & 0|1|2|3|4|5|6|7|8|9, \\
<\text{restnumber}> & \rightarrow & <\text{digit}> \mid <\text{digit}><\text{restnumber}>, \\
<\text{code}> & \rightarrow & <\text{sentence}> \mid <\text{code}>;<\text{sentence}>, \\
<\text{sentence}> & \rightarrow & <\text{assignment}> \mid <\text{loop}>, \\
<\text{id}> & \rightarrow & \texttt{X} <\text{number}>, \\
<\text{loop}> & \rightarrow & \texttt{while}\ \ <\text{id}> \neq 0\ \ \texttt{do}\ \ <\text{code}>\ \ \texttt{od}, \\
<\text{assignment}> & \rightarrow & <\text{id}> := 0 \quad \mid <\text{id}> := <\text{id}> \mid \\
 & & <\text{id}> := <\text{id}> +1 \quad \mid <\text{id}> := <\text{id}> -1\}
\end{array}
$$

}

(c) Para un programa P, con entrada $X1 = n$ que calcula la función $H(n)$ y tiene complejidad $T(n)$, dar una relación entre $H(n)$ y $T(n)$.

Solución.
Dado un programa P el cual calcula la función $H(n)$, con entrada $X1$, tiene complejidad $T(n)$, es decir, interpretando P como un algoritmo, $O(P) = T(n)$. Debido a la ambigüedad del enunciado, nos vemos obligados a considerar dos casos:

1. El programa P calcula única y exclusivamente la función $H(n)$, es decir, escribiendo dicho programa en EXWHILE:

   P = (H(n))

   Ahora bien, como $P = H(n)$, entonces $O(P) = O(H(n))$, por lo que $O(P) = T(n)$ y la complejidad temporal de P es $T(n)$.

   Veamos esto anterior para un ejemplo, en el que la función $H(n) = n!$, es decir, P es el programa que calcular el factorial de un número. Se puede comprobar que $T(n) = n$, o lo que es lo mismo, $O(P) = n$.

Esto es un claro ejemplo de que si estamos trabajando en este programa y tenemos una complejidad temporal superior; es decir, nuestro programa hace algunas iteraciones más, podemos concluir que hemos cometido algún error.

2. El programa P calcula en su interior la función $H(n)$, pero este puede realizar algunas operaciones más.

   En este caso, $P \neq H(n)$, pero $O(P) = T(n)$, por lo que $O(P) = T(n)$ por el propio enunciado. Solo que ahora, podría suceder que P tuviera bucles anidados que hicieran que la complejidad de P aumentara. Por ejemplo, sea P un programa que calcula $(n!)^n$, y $H(n) = n!$, la misma función del ejercicio anterior. Es claro que P calcula $H(n)$ en su interior.

   Sin embargo, se puede demostrar que la complejidad de P es $O(nlog(n))$, por tanto, la relación que se obtiene es:

   $$O(P) = T(n) \geq O(H(n))$$

2. **Ejercicio en la olimpiada matemática 2022**

   1. **Es la hora de comer en el zoo y los animales se ponen en cola a la espera de que abran el comedor. Pero hoy están tardando en abrir y el hambre aprieta. Los animales se han situado en una improvisada cola, cumpliendo el distanciamiento de metro y medio y nadie los ha ordenado para evitar que, como consecuencia del hambre, se coman unos a otros.**
   2. **En este zoo, un animal se puede comer a otro de un tamaño hasta D centímetros (cm) inferior. Así, si D = 2, un animal que mida 14 cm puede comerse a otro de 13 cm o 12 cm, pero no a animales de 11 cm o menos. No podría comerse a animales de su mismo tamaño o superior.**
   3. **Cuando un animal se come a otro, ocupa la posición en la cola del animal que ha sido devorado y el resto de la cola se compacta volviendo todos al distanciamiento de metro y medio.**
   4. **La naturaleza es sabia pero cuando tiene hambre pierde las formas, así que los animales se comerán unos a otros en el orden en el que finalmente queden menos animales vivos.**
   5. **Queremos saber cuántos animales sobrevivirán después de esta larga espera. Pensad en una solución por fuerza bruta. No hay que implementar la solución, bastará con describirla. ¿Cuánto tiempo se tardará en encontrar la solución si hay n animales?**

Solución.
FUERZA BRUTA

Existe una versión programada y vamos a proceder a explicarla

Partimos del mencionado array y del entero
$Animales[] = 7, 3, 5, 4$
$Alt = 2$

1. Recorremos todo el array
2. Por cada posición vemos si esta puede comer o no:
   a) Si puede comer, lo hace (se quita el comido del array) y se vuelve a llamar a esta función
   b) Si no, no se hace nada
3. El algoritmo se queda, de todas las devueltas, con la cadena de menor tamaño

$Complejidad = O(2^n)$

VORAZ

También existe una versión programada de esta.

1. Recorremos todo el array
2. Por cada posición vemos cuántos vecinos se puede comer cada uno.
3. El que pueda comer más será el elegido y se comerá a sus vecinos.
4. Repetimos hasta que nadie pueda comer más

$Complejidad = O(n^2)$

Puede encontrar la presentación del ejercicio 4, realizada en clase en este enlace. Además, puede encontrar los códigos del ejercicio 4 en nuestro repositorio de GitHub: `https://github.com/juanmagdev/AyC-Grupo-A1-04`