DIGITALHOUSE

Programación Web Full Stack

Alfabetización Digital III

Bienvenidos de nuevo a Alfabetización Digital, estos documentos gigantes que nos dan un poco de miedo pero también un montón de información útil.

En esta última entrega hablaremos sobre lógica proposicional y algoritmos, poniendo énfasis en qué cosas debemos pensar y cuáles son algunas herramientas que podemos utilizar para construir buenos algoritmos.

Lógica y Algoritmos

¿Qué es programar?

Ya habíamos definido que un **programa**, o software, es un conjunto de instrucciones u operaciones que se ejecutan en una computadora. Este conjunto de instrucciones tiene como objetivo resolver un problema particular.

"Programar es el proceso de escribir, probar y mantener un programa"

Para ser buenos desarrolladores hay que tener una buena comprensión del pensamiento lógico de programación y de la construcción efectiva de algoritmos.

¿Por qué aprender lógica?

Probablemente, no nos hayamos dado cuenta de lo mucho que la lógica está presente en nuestra vida cotidiana. La palabra "lógica" viene del griego "logos", que tiene múltiples traducciones: "razonamiento", "discurso" y "lenguaje". La lógica es tradicionalmente definida como el estudio de las leyes del pensamiento o el razonamiento correcto.
En este sentido, la lógica nos ayudará a interpretar correctamente los problemas y "pensar correctamente" las soluciones para los mismos.

Proposiciones, premisas y conclusiones

Veamos algunos términos de la *lógica proposicional*:

- Una **proposición** es una **oración declarativa** que puede ser ***verdadera*** o ***falsa***.
- Una **premisa** es una *proposición* que *se conoce o se asume* cómo ***verdadera***
- Una **conclusión** es una *proposición* que se **deriva mediante análisis lógico deductivo** en base a *dos o más premisas relacionadas*

Veamos un ejemplo:

- Si el tiempo está soleado, iremos a la playa. - Premisa
- Hoy está lloviendo - Premisa
- Por lo tanto, no iremos a la playa. - Conclusión

La última proposición es un análisis lógico deductivo (conclusión) basado en las dos primeras proposiciones (premisas).

Ejemplo 2:

- "Los jugadores de fútbol pueden tener la suerte de marcar un gol". - Premisa
- "En ese partido, el resultado fue de cero a cero". - Premisa
- "Por lo tanto, los jugadores no tuvieron suerte" .- Conclusión

Ejemplo 3:

- “Todo lo que fortalece la salud es útil”. - Premisa
- “El deporte fortalece la salud”. - Premisa
- “El atletismo es un deporte”. - Premisa
- “Por lo tanto, el atletismo es útil”. - Conclusión

Operadores Lógicos

A menudo, al trabajar con proposiciones, nos encontramos con algunas "conexiones" que pueden pasar desapercibidas en nuestra vida cotidiana, pero —cuando las observamos desde una perspectiva lógica— veremos lo importantes que son.
A continuación, veremos los principales operadores lógicos y cómo conectan las proposiciones.

Conjunción: operador “y”

De forma muy sencilla, el operador de conjunción “y” conectará dos o más proposiciones, añadiendo información que complementa la oración. Para que una afirmación se considere verdadera, ambas partes que la componen deben ser verdaderas.
Ejemplo:

"El alumno es participativo **y** muy estudioso".

Notar que por características del lenguaje español, en la frase anterior asumimos el “sujeto” de la segunda proposición. Aunque suene raro en nuestro idioma, el verdadero significado de la frase es:

“El alumno es participativo **y** *el alumno es* muy estudioso”

Esta distinción es importante porque en ***computación*** no se puede asumir el sujeto de la proposición, pero eso ya lo veremos más adelante.

Veamos otro ejemplo:

“Maria es muy diligente en las clases en vivo **y** Roberto es diligente en los estudios asincrónicos”

La presencia del conector “y” entre las dos proposiciones, hace que toda la oración sea verdadera si y sólo si, las dos oraciones son verdaderas.

Disyunción: operador “o”

A diferencia del operador anterior, el operador de disyunción “o” provoca la idea de separación entre dos o más proposiciones, donde entenderemos que puede ser verdadera la primera o la segunda.
En este caso, para que la oración completa se considere verdadera, es necesario que al menos una de las proposiciones que la componen sea verdadera.
Ejemplo:

“Maria fue al cine **o** al circo”

Analizando este ejemplo, tenemos dos situaciones que pueden darse de forma aislada: Maria pudo haber ido al cine o al circo. Para que la frase sea verdadera basta con que alguna de las dos proposiciones sean verdaderas, pudiendo también haber ido a ambos lugares.

Negación: operador “no”

Negar una proposición significa invertir el valor de verdad que la misma tenía. Si originalmente la proposición era verdadera y la negamos, se vuelve falsa y si originalmente era falsa, se vuelve verdadera.
Por ejemplo

Maria **no** fue al circo

Uniendo todo

Ejemplo:

“Maria salió de casa **y** fue al cine **o** al circo”.
“Maria **no** fue al circo”
“Por lo tanto, Maria fue al cine.”

En este ejemplo podemos ver como construir una serie de premisas utilizando operadores lógicos y llegar a una conclusión en base a las mismas.

Algoritmos

¿Qué es un algoritmo?

Cuando se habla de algoritmos es habitual relacionarlos a tecnologías complicadas y difíciles de entender, pero muy distinto a esto, los algoritmos están presentes en todo lo que hacemos.

"Un **algoritmo** es una **secuencia**, *finita* y *rigurosa*, de **instrucciones** típicamente usada para **resolver una clase específica de problemas**."

Veamos algunos ejemplos de algoritmos que seguramente conozcan:

- Receta para hacer una torta
- Manual de instrucciones para ensamblar un escritorio
- Viaje desde su casa hasta su trabajo.

En nuestra vida cotidiana, para realizar cualquier tipo de tarea, llevamos a cabo acciones organizadas en una secuencia ordenada. La programación de software es muy similar a estas actividades.

Entonces, sabiendo qué problema debe resolver el programa, analizamos e identificamos todos los pasos necesarios para llegar a esa solución y, sólo entonces, construimos dicha solución en algún lenguaje de programación puntual.

Pensando en algoritmos

Es algo muy común que un principiante tenga como tarea resolver un algoritmo y que esa tarea le parezca completamente insuperable. Si esto te sucedió, ¡no te preocupes! Es completamente normal y le pasa a cualquiera que esté arrancando a hacer alguna tarea por primera vez. O incluso segunda, tercera, o cuarta vez.
Lo importante es tener un buen proceso para comprender el problema, desmenuzarlo en partes más sencillas de resolver, e identificar las actividades que vamos a tener que realizar para resolverlas.

Crear algoritmos para computadoras **no se trata de memorizar**, **no se trata de matemática**, **no se trata de las particularidades del lenguaje que estemos usando**.
Crear algoritmos para computadoras se trata de **poder explicarle a una computadora totalmente ignorante**, que no tiene ningún conocimiento, ni ningún contexto, **cómo resolver un problema en particular**.
Simplemente, **significa poder expresar en términos que la computadora pueda entender, qué tiene que hacer**. Ese es el único obstáculo que deben superar.
Si pudieron aprender a hablar en un lenguaje natural y expresar ideas tan complejas entre personas como las que manejamos día a día, definitivamente van a poder aprender un lenguaje de programación y a escribir programas. **Los lenguajes de programación son mucho más simples que los lenguajes naturales**.

Algoritmos en frases coloquiales

Ya hemos visto diagramas de flujo que resultan muy útiles para construir algoritmos de una manera abstracta. Estos nos van a dar una idea de cómo se van a estructurar nuestros algoritmos. Sin embargo, desde ese punto hasta el código en un lenguaje de programación todavía hay un gran salto.

Una herramienta posible es antes de generar el

Cuando sea la hora de escribir en un lenguaje de programación real, te recomendamos que **no arranques con código** del lenguaje, sino **con comentarios usando pseudocódigo** o **simplemente oraciones en tu lenguaje natural**.
Esto te va a ayudar a fraccionar los problemas en tareas más sencillas y a definir realmente cómo serán los pasos para cada una
A medida que te vayas sintiendo más cómodo con los algoritmos y el lenguaje de programación particular, podrás eventualmente dejar de usar esta herramienta.

Datos y Pasos

Siempre que pensamos en algoritmos tenemos que tener dos cosas fundamentales en cuenta.

¿Cuáles son mis datos y qué tipos de datos son?

En otras palabras, ¡nuestras variables y sus tipos de datos! No hay muchos tipos de datos, repasemos un poquito:

Tipos de datos primitivos

- **números** (enteros, reales, imaginarios), ej: 42
- **texto** o cadenas de caracteres, strings, ej: "`un texto`"
- **valores de verdad**, booleans, ej: `true`

Tipos de datos no-primitivos

- **Listas de elementos**, arrays, ej: `[5,"hola",falso]`
- **Estructuras** que asocien alguna llave con algún valor, objetos.
  Ej: `{ unaLLave="unvalor", otraLlave=22 }`
- **Funciones**. Decimos que es un tipo de dato porque una variable puede guardar funciones.

Estos tipos de datos existen en todos los lenguajes de programación, y se dice que son ortogonales, es decir, pocos, simples y muy combinables entre ellos.
No es importante la sintaxis de ellos (es decir cómo los escribimos), sino el concepto. "Tengo un tipo de dato para representar texto, otro para números". Y cada uno tiene "comportamiento" distinto. "Puedo agregar un texto a otro", "puedo multiplicar dos números", "puedo insertar elementos en una lista", "puedo agregar o eliminar llaves de un objeto".
Cuando tengamos una variable, nunca perdamos de vista qué tipo de dato es y a través de ese tipo de dato, qué operaciones puedo realizar sobre ella.

¿Cómo son los pasos?

Necesito entender mi problema, para entender cómo son mis tipos de datos iniciales y cómo son mis tipos de datos resultantes. Al entenderlo, voy a poder generar los pasos para explicarle a la computadora cómo debe generar esa transformación. ¡Esto es escribir algoritmos!

Dentro de los pasos, hay dos conceptos fundamentales que necesitamos entender: Condicionales y ciclos. Repasemos!

Condicionales

También conocidos como bifurcaciones o branching, son bloques de mi algoritmo que se ejecutarán condicionalmente. Volviendo a los ejemplos de cocina:

```
if (cuchara.estaSucia){
      cuchara.revolver(bowl)
}else{
      tenedor.revolver(bowl)
}
```

Los condicionales siempre actúan sobre alguna proposición (que recordemos que pueden ser verdaderas o falsas). Si la proposición es verdadera, se ejecuta el primer bloque. Si la proposición es falsa, se ejecuta el bloque del "sino". **Nunca** se pueden ejecutar **ambos bloques**, ya que una proposición no puede ser verdadera y falsa al mismo tiempo.

Ciclos

También conocidos como bucles o loops, son bloques de mi algoritmo que yo quiero que se ejecuten muchas veces. Estos operan sobre alguna variable y tienen alguna proposición definida que en caso de volverse falsa detienen el ciclo.

Por ejemplo:

```
for (let taza=200; taza > 0; taza=taza-50){
      bowl.agregar(50)
      bowl.revolver()
}
```

Resolviendo problemas

No importa cuánto leamos o cuantos videos miremos, o cuantos tutoriales copiemos y peguemos. Los materiales teóricos son importantes, pero ¡La única manera de aprender programación es practicando! Y la mejor manera de practicar algoritmos es planteándose problemas e intentar resolverlos.

Veamos un ejemplo de un algoritmo interesante que seguramente muchos de ustedes ya estén familiarizados, y veamos cómo resolverlo paso a paso.

Factorial

Queremos implementar una función “Obtener Factorial” que reciba un número entero y devuelva el factorial de ese número.
Esta operación la conocemos, pero incluso aunque no la conozcamos, sabemos del enunciado cómo se verá mi función:

```
1  function ObtenerFactorial (numero){
2
3  }
```

Para calcular un factorial, debemos multiplicar todos los números enteros positivos desde 1 hasta el entero del cuál queremos calcular el factorial. En este caso “número”. ¿Pero cómo sabemos qué número nos van a pedir? Podríamos usar condicionales de una manera ingenua:

```
1  function ObtenerFactorial (numero){
2    if (numero==1){
3        return 1
4    }
5    if (numero==2){
6        return 1*2
7    }
8    if (numero == 3){
9        return 1*2*3
10   }
```

```
11    (...)
12  }
```

Pero rápidamente nos damos cuenta que estaríamos escribiendo este programa por el resto de la eternidad. Simplemente no sabemos qué número nos van a mandar.

Y no necesitamos saberlo. Si miramos detenidamente el algoritmo anterior, podemos descubrir un patrón. Arrancamos desde el número que nos dieron y multiplicamos por todos los números hacia abajo hasta 1. Este patrón lo descubrimos pensando en el algoritmo que nosotros teníamos que hacer con nuestras manos para escribir esa solución ingenua.

Empecemos de nuevo. Sabemos que “número” es un entero. Y debería ser positivo. Partamos desde esa premisa y construyamos una conclusión:

```
1  function ObtenerFactorial (numero){
2    si (numero <= 1){
3        return 1
4    }
5  }
```

Perfecto, resolvimos un caso de factorial. Mi función ahora se comporta como debería para todos los números de 1 hacia infinito negativo. A esto lo llamamos un caso base. Un buen patrón para resolver algoritmos es fraccionar nuestra solución en una serie de casos base y atacar cada uno por separado.

Ahora, ¿Qué pasa si “número” es mayor a 1? Deberíamos empezar desde 2 y multiplicar ese valor por el siguiente número hasta llegar al número que necesitamos. Esto es claramente un ciclo. Un ciclo tiene un punto de inicio, un punto de corte, y un incremento. No importa en qué lenguaje, esto es así. Miremos cómo podría ser ese ciclo:

```
1  función ObtenerFactorial (entero número){
2    si (número <= 1){
3        devolver 1
4    }
5    for (let i=2; i<=n; i++){
```

```
6       //i irá desde 2 hasta n
7    }
8  }
```

“i” será el valor que irá incrementando, y representará los diferentes operandos de la multiplicación que utilizaremos. Supongamos que “número=4”. Entonces el calculo seria:

```
!4 = 1*2*3*4 = 24
```

Desmenuzado, también puede verse así:

```
!4 = ((2)*3)*4 = (6)*4 = 24
```

Es decir que el resultado de la primera multiplicación debe multiplicarse por el siguiente número. Si lo tuviéramos que transformar a código, deberíamos tener una variable que pueda almacenar ese dato “intermedio” del resultado de la multiplicación anterior. A este tipo de variables las llamamos “acumuladores”.

¿Y qué valor tendrá que tener inicialmente esta variable? ¡Debería empezar en “1”!

Poniendo todo esto en práctica, podría quedarnos algo así:

```
1  function ObtenerFactorial (numero){
2    if (numero <= 1){
3        return 1
4    }
5    let resultado = 1
6    for (let i=2;i<=numero;i++){
7        resultado = i*resultado
8    }
9    //Al finalizar el ciclo, “resultado” tendrá el valor final
10 }
```

Ahora nos queda un único paso más. ¿Se les ocurre qué es? ¡Claro, devolver la variable resultado con el valor calculado!

Resultado Final:

```
function ObtenerFactorial (numero){
  if (numero <= 1){
      return 1
  }
  let resultado = 1
  for (let i=2;i<=numero;i++){
      resultado = i*resultado
  }
  return resultado
}
```

Más ejercicios

Les dejamos más ejercicios como para que pongan esto en práctica. Al final del documento encontrarán algunas posibles soluciones, pero **recuerden que nunca hay una única manera de resolver algo**. Prueben sus soluciones en Node, comparenlas con las de sus compañeros y discutan por qué algunas son mejores que otras.
No importa cómo escriban los algoritmos, estos ejercicios son para practicar lógica y pensar en algoritmos.

1- Invertir texto

Escribir el algoritmo de la función **InvertirTexto** que reciba una variable "**oración**" de tipo **cadena de caracteres** (es decir texto) y que devuelva ese valor con los caracteres del texto en orden inverso.

2- Primera palabra

Escribir el algoritmo de la función **"PrimeraPalabra"** que reciba una variable "**oración**" de tipo **cadena de caracteres** y que devuelva la primera palabra de dicha oración. Una palabra es cualquier cadena de caracteres hasta un espacio (el espacio es un carácter, y no es parte de la palabra).

3 - Última palabra

Escribir el algoritmo de la función **"UltimaPalabra"** que reciba una variable "**oración**" de tipo **cadena de caracteres** y que devuelva la última palabra.

4- Lista de palabras

Escribir el algoritmo de la función **“Lista de palabras”** que reciba una variable “oración” de tipo **cadena de caracteres** y que devuelva una lista con cada una de las palabras.
Por ejemplo, dada la oración “esta es una frase muy interesante”, debería devolver la lista: [“esta”,”es”,”una”,”frase”,”muy”,”interesante”]
¿Cómo cambiarías tu algoritmo si necesitamos solo la cantidad de palabras en la oración?

5- Palabra más larga

Escribir el algoritmo de la función **“PalabraMásLarga”** que reciba una variable “oración” de tipo **cadena de caracteres** y que devuelva la palabra más larga de la oración. Si dos o más palabras comparten el tamaño máximo, devolver la última.
¿Cómo cambiarías tu algoritmo para que en el caso de haber muchas palabras del mismo tamaño en vez de devolver la última devuelva la primera?

1- Solución: Invertir texto

```
function InvertirTexto(oración) {
   //Obtengo el largo de la oración para poder recorrerla
   const largo = oración.length;
   //Inicializo la oración invertida
   let oraciónInvertida = "";
   //Ciclo para recorrer cada posición de la oración
   // i va de largo-1 a 0
   for (let i = largo - 1; i >= 0; i--) {
       //Agrego los caracteres desde el final de oración
       //Al principio de oraciónInvertida
       oraciónInvertida += oración[i];
   }
   //Devuelvo la oraciónInvertida
   return oraciónInvertida;
}
```

2- Solución: Primera palabra

```
function PrimeraPalabra(oración) {
   //Obtengo el largo de la oración para poder recorrerla
   const largo = oración.length;
   //Inicializo la variable que contendrá la primer
   // palabra
   let palabra = "";
   //Ciclo para recorrer cada posición de la oración
   // i va de 0 a el largo-1
   for (let i = 0; i < largo; i++) {
     //Obtengo el caracter en la posición i del texto
     const caracter = oración[i];
     //Si el caracter es un espacio
     if (caracter == " ") {
       //Devuelvo la primer palabra
       return palabra;
     } else {
       //Agrego el caracter a palabra
       // ya que es parte de la palabra
       palabra += caracter;
     }
   }
   //Devuelvo la palabra en el
   // caso de que todo el texto sea solo una palabra
   return palabra;
}
```

3- Solución: Última palabra

```
function UltimaPalabra(oración) {
   //Obtengo el largo de la oración para poder recorrerla
   const largo = oración.length;
   //Inicializo la variable que contendrá la palabra
   // que voy leyendo
   let palabraActual = "";
   //Ciclo para recorrer cada posición de la oración
   // i va de 0 a el largo-1
   for (let i = 0; i < largo; i++) {
     //Obtengo el caracter en la posición i del texto
     const caracter = oración[i];
     //Si el caracter es un espacio
     if (caracter == " ") {
       //Reinicio la palabra
       palabraActual = "";
     } else {
       //Agrego el caracter a palabra ya que es parte
       // de la palabra
       palabraActual += caracter;
     }
   }
   //Devuelvo ultima palabra que capturé
   return palabraActual;
}
```

4- Solución: Lista de palabras

```
function ListaDePalabras(oración) {
   //Obtengo el largo de la oración para poder recorrerla
   const largo = oración.length;
   //Inicializo la variable que contendrá todas
   // las palabras
   let palabras = [];
   //Inicializo la variable que contendrá la
   // palabra que voy leyendo
   let palabraActual = "";
   //Ciclo para recorrer cada posición de la oración
   // i va de 0 a el largo-1
   for (let i = 0; i < largo; i++) {
       //Obtengo el caracter en la posición i del texto
       const caracter = oración[i];
       //Si el caracter es un espacio
       if (caracter == " ") {
           //Agrego la palabra capturada hasta el momento
           palabras.push(palabraActual);
           //Reinicio la palabra
           palabraActual = "";
       } else {
           //Agrego el caracter a palabra ya que es
           // parte de la palabra
           palabraActual += caracter;
       }
   }
   //Devuelvo la lista de palabras
   return palabras;
}
```

5- Solución: Palabra más larga

```
function PalabraMásLarga(oración) {
   //Obtengo el largo de la oración para poder recorrerla
   const largo = oración.length;
   //Inicializo la variable que contendrá la palabra
   // más larga
   let palabraMásLarga = "";
   //Inicializo la variable que contendrá la palabra
   // que voy leyendo
   let palabraActual = "";

   //Ciclo para recorrer cada posición de la oración
   // i va de 0 a el largo-1
   for (let i = 0; i < largo; i++) {
       //Obtengo el caracter en la posición i del texto
       const caracter = oración[i];
       //Si el caracter es un espacio
       if (caracter == " ") {
           if (palabraActual.length >= palabraMásLarga.length) {
               palabraMásLarga = palabraActual;
           }
           //Reinicio la palabra
           palabraActual = "";
       } else {
           //Agrego el caracter a palabra ya que es
           // parte de la palabra
           palabraActual += caracter;
       }
   }
   //Devuelvo la palabra más larga
   return palabraMásLarga;
}
```