Guía del curso

Módulo 2 Diseño de Algoritmos

2.1 Algoritmos y su Importancia

2.1.1 Definición y características de un algoritmo

Definición de Algoritmo

Un algoritmo es un conjunto finito de instrucciones o pasos que describe un proceso o conjunto de operaciones a seguir para realizar una tarea o resolver un problema específico. Los algoritmos son esenciales en el campo de la programación, ya que proporcionan una secuencia clara y lógica de pasos que, cuando se siguen correctamente, conducen a la solución de un problema o a la ejecución de una tarea deseada.

Características de un Algoritmo

- **Precisión:** Un algoritmo debe ser preciso y no ambiguo. Cada paso debe estar definido de manera clara y sin posibilidad de interpretaciones múltiples.
- **Finito**: El algoritmo debe tener un número finito de pasos o instrucciones. No puede ser infinito, ya que debe ser ejecutable en un tiempo razonable.
- **Eficacia y eficiencia:** Un algoritmo debe ser eficaz, es decir, debe resolver el problema para el cual fue diseñado. Además, debe hacerlo en un tiempo razonable y utilizando recursos necesarios.
- **Entrada y Salida:** Todo algoritmo tiene datos de entrada y produce resultados. Los datos de entrada son la información con la que el algoritmo opera, y los resultados son la solución al problema o el resultado de la tarea.
- **General:** Un algoritmo debe ser lo suficientemente general como para aplicarse a todas las instancias del problema para el cual fue diseñado. Debe ser capaz de manejar diferentes conjuntos de datos y condiciones.
- **Independencia de la Máquina o del lenguaje de programación:** Un buen algoritmo debe ser independiente de la máquina o del lenguaje de programación específico. Debería poder implementarse en diferentes entornos y lenguajes.
- **Claridad y Legibilidad:** La claridad y la legibilidad son esenciales en un algoritmo. Debe ser comprensible por otros programadores y personas que revisen el código.

- **Repetibilidad:** Un algoritmo debe ser repetible. Si se sigue el mismo conjunto de pasos con la misma entrada, siempre se producirá la misma salida.

Estas características aseguran que un algoritmo sea una herramienta efectiva para resolver problemas y realizar tareas de manera sistemática y eficiente. La capacidad de diseñar y comprender algoritmos es fundamental en la programación y la resolución de problemas computacionales.

Práctica 2.1. Elaborar algoritmo y programa en Python para calcular la fuerza de atracción de dos cuerpos si se sabe que su fórmula es F = G * (masa1 * masa2) / distancia ^2. Esta práctica tiene varios datos de entrada y produce una salida.

Ejercicio 2.1. Elaborar algoritmo y programa en Python para determinar la temperatura en grados Farenheit y Kelvin si se proporciona en grados centigrados (Un dato de entrada y dos salidas).

Ejercicio 2.2. Elaborar algoritmo y programa en Python para determinar conversiones (peso, distancia, divisas, etc.) (Un dato de entrada y varias salidas).

2.1.2 Diseño de algoritmos para resolver problemas específicos

Diseñar algoritmos es el proceso de crear un conjunto ordenado y preciso de instrucciones o reglas que describen cómo realizar una tarea o resolver un problema específico. Estos algoritmos son esenciales en la programación y la resolución de problemas computacionales. El proceso de diseñar algoritmos consiste en:

- **Entender el Problema:** Antes de diseñar un algoritmo, es crucial comprender completamente el problema que estás intentando resolver. Esto implica analizar los requisitos del problema y entender cuál es el resultado deseado.

- **Identificar los Datos de Entrada:** Determinar qué datos o información serán proporcionados al algoritmo al inicio de su ejecución. Estos son los valores iniciales que el algoritmo utilizará para realizar sus cálculos o acciones.

- **Definir los Pasos o Instrucciones:** Descomponer el problema en pasos más pequeños y manejables. Define las operaciones o acciones específicas que el algoritmo debe realizar en cada uno de estos pasos. La claridad y la precisión son fundamentales en esta etapa.

- **Considerar Estructuras de Control:** Incorporar estructuras de control como bucles (loops) y condicionales (if-else) para manejar la repetición de pasos o tomar decisiones basadas en ciertas condiciones.

- **Evaluar la Eficiencia:** A medida que se diseña el algoritmo, considerar la eficiencia del mismo. Esto implica evaluar cuántos recursos computacionales (tiempo, memoria)

consume el algoritmo y si puede manejar grandes conjuntos de datos de manera eficiente.

- **Pensar en la Generalidad:** Diseña el algoritmo de manera que sea aplicable a una amplia variedad de situaciones o casos. Un buen algoritmo no debería estar limitado a un conjunto específico de datos o condiciones.

- **Utilizar Pseudocódigo o Diagramas de Flujo:** Antes de escribir el código en un lenguaje de programación específico, es útil expresar el algoritmo de manera más cercana al lenguaje natural. Esto se puede hacer mediante el uso de pseudocódigo o diagramas de flujo.

- **Probar y Depurar:** Una vez diseñado el algoritmo, es importante realizar pruebas exhaustivas. Asegúrarse de que funcione correctamente para diversos casos de prueba y corregir cualquier error (depuración) que pueda surgir.

- **Documentar:** Documentar el algoritmo de manera clara. Esto puede incluir comentarios en el código (si es un algoritmo implementado en un lenguaje de programación) o descripciones detalladas si se está utilizando pseudocódigo o diagramas de flujo.

- **Optimizar (si es necesario):** Si el algoritmo funciona pero hay oportunidades para mejorar su eficiencia, considera realizar optimizaciones. Sin embargo, la optimización no siempre es necesaria y debe equilibrarse con la legibilidad y manteniento del código.

Un buen diseño de algoritmos puede marcar la diferencia en términos de eficiencia, mantenimiento y escalabilidad de un sistema o programa.

Ejemplo 2.1. Diseñar el algoritmo para el planteamiento del ejercicio 1.10. El algoritmo se muestra en forma de texto.

```
// Guardar con valores False o True ante los planteamientos presentes.
// Se considerarán los siguientes accesos:
// Acceso a dama
// Acceso a caballero
// Acceso a menor
Inicio
        Repetir para cada acceso
                En apariencia, es mayor de edad?
                Si no, solicitar credencial INE
                Verificar edad
                Si es menor de edad
                        Buscar en lista especial
                        Si esta, permitir acceso
                        Sino (de lo contrario) negar acceso
                De lo contrario  // Es mayor de edad.
                Es dama?
                Si, solicitar si trae acompañante
                        Es caballero?
```

Si, permitir acceso (ambos)
// Si fuera otra dama o menor de edad, se considera sin acompañante.
Sino trae acompañante
Solicitar si desea ingresar sola // Esta de más preguntar
Si es afirmativo permitir acceso
De lo contrario negar acceso
Preparar nuevo acceso
Hasta proporcionar clave de terminación
Fin

2.2 Técnicas de Diseño de Algoritmos

2.2.1 Técnicas de diseño de algoritmos (Dividir y conquistar, de regreso <backtrack>, uso de patrones y analogías)

Existen varias técnicas de diseño de algoritmos que los programadores (diseñadores de algoritmos) utilizan para abordar diferentes tipos de problemas de manera efectiva. Algunas de las más comunes son:

- **Divide y Vencerás (Divide and Conquer):** Divide el problema en subproblemas más pequeños, resuelve cada subproblema de manera independiente y luego combina las soluciones para obtener la solución final.

Ejemplo 2.2. El algoritmo de ordenación rápida (quicksort) y el algoritmo de búsqueda binaria utilizan divide y vencerás.

- **Programación Dinámica:** Divide un problema en subproblemas más pequeños y resuelve cada subproblema solo una vez, almacenando las soluciones ya calculadas para evitar redundancias.

Ejemplo 2.3. El algoritmo de la mochila (knapsack problem) y el algoritmo de Floyd-Warshall para encontrar caminos más cortos en grafos utilizan programación dinámica.

- **Algoritmos Voraces (Greedy Algorithms):** Toma decisiones locales óptimas en cada paso con la esperanza de llegar a una solución global óptima.

Ejemplo 2.4. El algoritmo de Kruskal para encontrar el árbol de expansión mínima de un grafo utiliza el algoritmo voraz.

- **Algoritmos de Retroceso (Backtracking):** Explora sistemáticamente todas las posibles soluciones para encontrar la mejor.

Ejemplo 2.5. El problema de las N reinas y el problema del viajero (traveling salesman problem) utilizan algoritmos de retroceso.

- **Algoritmos Voraces Incrementales (Incremental Greedy Algorithms):** Similar a los algoritmos voraces, pero toma decisiones óptimas a medida que avanza en lugar de anticiparse.

Ejemplo 2.6. El algoritmo de Dijkstra para encontrar el camino más corto en grafos ponderados utiliza el algoritmo voraz incremental.

- **Algoritmos de Búsqueda Local (Local Search Algorithms):** Encuentra soluciones mejorando iterativamente una solución inicial mediante movimientos locales.

Ejemplo 2.7. El algoritmo de búsqueda local en el problema del viajero.

- **Ramificación y Acotación (Branch and Bound):** Genera y explora sistemáticamente todas las soluciones posibles mediante la creación de un árbol de soluciones, eliminando las ramas que no pueden conducir a una solución óptima.

Ejemplo 2.8. Resolución del problema del viajero con el algoritmo de ramificación y acotación.

- **Uso de Patrones:** Identificar patrones recurrentes en problemas similares y aplicar soluciones que hayan funcionado en situaciones anteriores.

Ejemplo 2.9. Si se ha resuelto un problema antes y si se encuentra un problema similar, se puedes aplicar el mismo enfoque.

- **Uso de Analogías:** Comparar el problema actual con situaciones o problemas conocidos para encontrar similitudes y aplicar estrategias que hayan funcionado en el pasado.

Ejemplo 2.10. Si se está abordando un problema (optimización en una red), se puede hacer una analogía con el flujo máximo en un grafo.

La elección de la técnica adecuada depende del tipo de problema que se este abordando y de las restricciones específicas del problema. Estas técnicas suelen utilizarse a menudo en combinación y de la riqueza del conocimiento acumulado, así como la creatividad del programador.

Ejercicio 2.3. Realiza una investigación en la web para tener una idea de los problemas planteados en los ejemplos de las diferentes técnicas de diseño.

2.2.2 Algoritmos representados de manera textual

Los algoritmos se pueden representar de manera textual utilizando diferentes notaciones dependiendo de la preferencia del diseñador o la convención de la comunidad de programadores dentro de una empresa o corporación. Cuando se habla de manera textual es porque se desea algo escrito, ya que podría hacerse de manera verbal donde se comenta los pasos a seguir que conforman el algoritmo. La manera texual puede ser una *redacción ordinaria o natural* donde se indique la secuencia de pasos. Otra manera textual es mediante *pseudocódigo*, el cual utiliza partes (código) de un lenguaje de programación y partes de redacción ordinaria, de ahí el nombre “código falso”. Otra manera más de representar un algoritmo de manera textual es utilzando una representación gráfica conocida como *diagrama de flujo de datos* (DFD). Cuando el programador ya posee cierta experiencia podría omitir los algoritmos textuales y pasar directamente al lenguaje de programación, colocando ya en código de programación el algoritmo que posee de manera mental (algunas veces lo va comentatado para si mismo a paa algún colega).

El pseudocódigo y los diagramas de flujo de datos son muy útiles para cuando se desea *describir la lógica de un algoritmo* de manera clara sin preocuparte por la sintaxis específica de un lenguaje de programación.

En muchas ocaciones como parte de la documentación del programa se requerirá plasmar los algoritmos utilizados y es aquí donde también se requiere su elaboración,

Ejemplo 2.11. Como ejemplo de un algoritmo mediante una redacción ordinaria o natural es el ejemplo 1.1 que muestra los pasos a seguir para determinar la suma de dos números, el cual se transcribe:

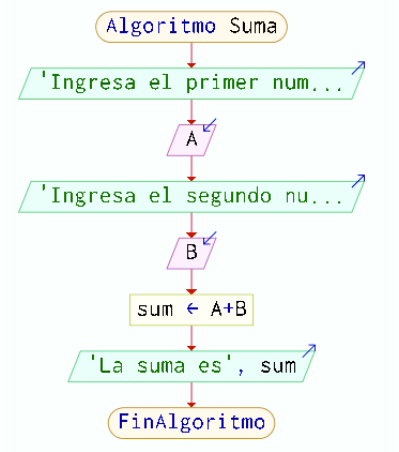
Algoritmo Suma
Inicio
Pedir al usuario que ingrese el primer número (A)
Pedir al usuario que ingrese el segundo número (B)
Suma = A + B
Mostrar "La suma es:", Suma
Fin

Ejemplo 2.12. Utilizando el mismo algoritmo de suma de dos números, pero ahora mediante pseudocódigo, el cual no sigue reglas estrictas de sintaxis, se vería como sigue:

Algoritmo *Suma*
Inicio
Escribir "Ingrese el primer número (A)"
Leer A
Escribir "Ingrese el segundo número (B)"
Leer B
Suma ← A + B
Escribir "La suma es:", Suma
Fin

Las leyendas leer y escribir se utilizan para obtener los datos por parte del usuario y mostrar los resultados respectivamente. De este pseudocódigo, el programador ya podría pasarlo a codificar a su lenguaje de programación de preferencia.

Ejemplo 2.13. Utilizando el mismo algoritmo de suma de dos números, pero ahora mediante diagrama de flujo de datos, elaborado con el programa PSeInt, se vería como sigue:



Ejemplo 2.14. Utilizando el mismo algoritmo de suma de dos números, pero ya convertido en el lenguaje de programación de Python, desarrollado en la práctica 1.3 y como se menciono, podría elaborarse directamente de la representación mental, se transcribe:

```
# Practica_1_3_Suma.py
# Aprendelo, November 4, 2023
#
# Suma dos números.

A = 10  # Valor predeterminado.
B = 20  # Valor predeterminado.

print("Ingresa el primer número:")
A = int(input())
print("Ingresa el segundo número:")
B = int(input())
suma = A + B
print("La suma es",suma)
```

2.3 Representación de Algoritmos mediante Diagramas de Flujo

2.3.1 Uso de diagramas de flujo para visualizar algoritmos.

Los diagramas de flujo son una herramienta gráfica utilizada en programación y diseño de algoritmos para representar de manera visual el flujo de control de un proceso. Estos diagramas son especialmente útiles para ilustrar la lógica de un algoritmo de manera clara y comprensible. Algunas pautas y ventajas sobre el uso de diagramas de flujo son:

Pautas para Crear Diagramas de Flujo Efectivos

- **Simplicidad:** Mantener el diagrama simple y fácil de entender. Evitar la complejidad innecesaria.
- **Consistencia:** Utilizar símbolos y formas de manera consistente a lo largo del diagrama.
- **Claridad:** Asegúrarse de que el diagrama sea claro y fácil de seguir. Etiquetar adecuadamente los símbolos y las flechas.
- **Orden Lógico:** Organizar los símbolos en un orden lógico que refleje el flujo de control del algoritmo.
- **Minimizar las Líneas Cruzadas:** Evitar cruces innecesarios de líneas para mantener la claridad del diagrama.

Ventajas de los Diagramas de Flujo

- **Visualización Clara:** Los diagramas de flujo proporcionan una representación visual clara de la lógica del algoritmo.
- **Facilitan la Comunicación:** Son útiles para comunicar la lógica del programa a personas no familiarizadas con la programación.
- **Análisis y Depuración:** Ayudan en el análisis y la depuración del algoritmo antes de su implementación.

2.3.2 Símbolos y convenciones en diagramas de flujo

Componentes Comunes de un Diagrama de Flujo:

- **Inicio/Fin:** Se representa con un óvalo y marca el comienzo y el final del algoritmo.
- **Leer:** Se representa con un trapecio y representa la lectura de datos.

- **Escribir:** Se representa con un rectángulo con el lado inferior ondulado y representa la operación escritura de resultados.

- **Proceso:** Se representa con un rectángulo y representa una operación o acción específica.

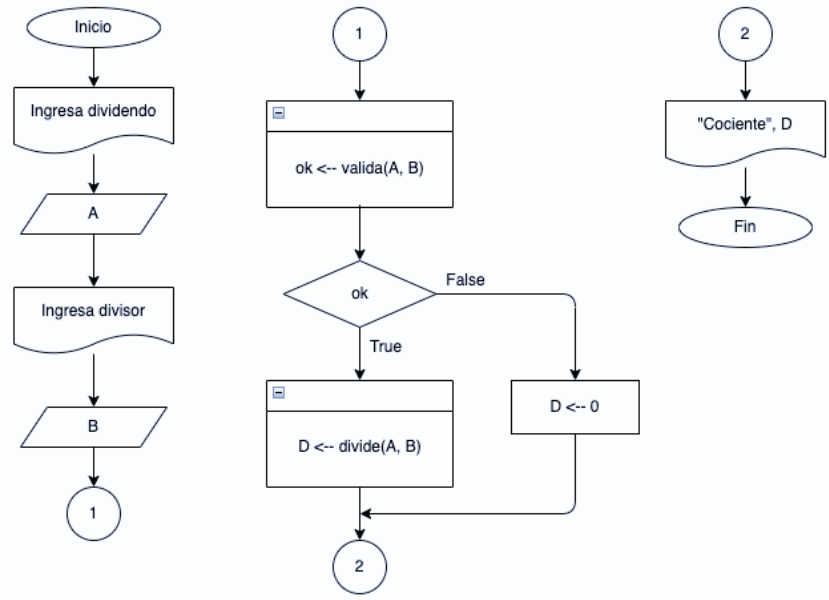
- **Rutina:** Se representa con un rectángulo con una línea horizontal adicional y representa la llamada a una rutina (procedimiento o función).

- **Decisión:** Se representa con un rombo y se utiliza para representar una decisión. La línea de salida depende de la condición (por ejemplo, verdadero o falso).

- **Conector:** Se utiliza para conectar partes del diagrama que están en diferentes páginas o secciones. Suele ser un círculo para la misma página y un como cuadrado con cinco lados para diferente página.

- **Flecha de Flujo:** Indica la dirección del flujo del control o de los datos, mostrando el orden en que se ejecutan las operaciones.

Ejemplo 2.15. Desarrollar un algoritmo mediante diagrama de flujo de datos que realiza la división de dos números siempre y cuando la validación de ellos sea correcta de lo contrario se mostrará cero. El diagrama, elaborado con el programa VS code con la extension Drawio Integration, incluye los componentes más comunes:



Herramientas que te permiten crear diagramas de flujo

Existen varias herramientas que te permiten crear diagramas de flujo y, en algunos casos, incluso ejecutar algoritmos directamente en el entorno de desarrollo. Algunas opciones son:

- **Draw.io:** Draw.io es una herramienta en línea que te permite crear diagramas de flujo de manera sencilla. Aunque no permite la ejecución de algoritmos, es excelente para la visualización y la planificación.

- **Lucidchart:** Similar a Draw.io, Lucidchart es una plataforma en línea para la creación de diagramas de flujo y otros diagramas. Ofrece una interfaz intuitiva y opciones de colaboración.

- **Visual Studio Code con Extensiones:** Si se esta familiarizado con Visual Studio Code, se puede utilizar para crear diagramas de flujo con extensiones como "Draw.io Integration". Además, puedes ejecutar fragmentos de código directamente en Visual Studio Code.

- **Jupyter Notebooks:** Jupyter Notebooks es una herramienta popular en el ámbito de la ciencia de datos. Se puede combinar celdas de texto explicativo con celdas de código ejecutable, lo que permite crear y ejecutar algoritmos junto con la documentación visual.

- **CodeSignal (Algoritmos en Línea):** Plataformas en línea como CodeSignal permiten escribir algoritmos y ejecutarlos en el navegador. Aunque no generan diagramas de flujo, son útiles para probar y depurar código.

- **PSeInt:** PSeInt es una herramienta que te permite crear algoritmos en pseudocódigo y su correspondiente diagramas de flujo de datos de manera sencilla. Permite la ejecución de los algoritmos en su propio lenguaje (ambiente de desarrollo). También permite exportar el algoritmo en diferentes lenguajes de programación.

En general, los diagramas de flujo son herramientas visuales para la planificación y la comunicación, y no suelen ejecutar algoritmos directamente. La ejecución de algoritmos suele llevarse a cabo en entornos de desarrollo específicos para un lenguaje de programación determinado.

Práctica 2.2. Descargue PSeInt del sitio SourceForge (https://sourceforge.net/projects/pseint/) e instale esta herramienta en su computadora.

Práctica 2.3. Una vez instalado el programa PSeInt, elabore el algoritmo mostrado en el ejemplo 2.13. Vea ambas opciones: en pseudocódigo y en diagrama. Ejecute el algoritmo y observe su ejecución.

Práctica 2.4. Exporte el algoritmo (codificarlo) en el lenguaje Python y compare con la versión mostrada en el ejemplo 2.14.

Práctica 2.5. Instale la extensión Drawio integration en su VS code.

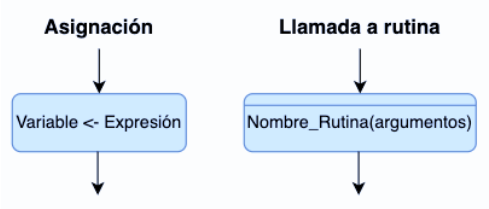
Práctica 2.6. Una vez instalada la extensión, elabore el algoritmo mostrado en el ejemplo 2.15.

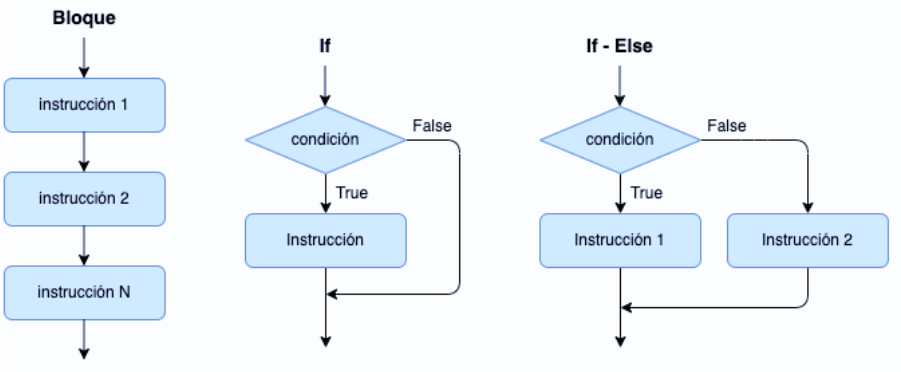
Diagramas de Flujo para las estructuras de control

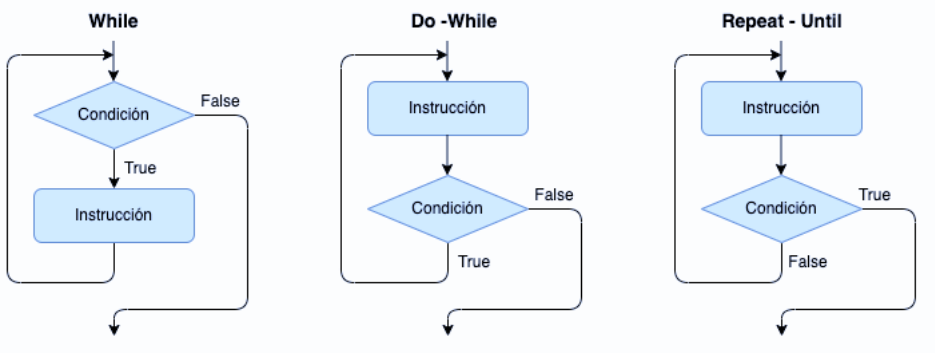
Como se menciono con anterioridad, un algoritmo es una secuencia de pasos para resolver un problema. Al representar el algoritmo en un diagrama de flujo de datos, cada paso es en sí una instrucción. Las instrucciones suelen clasificarse en:

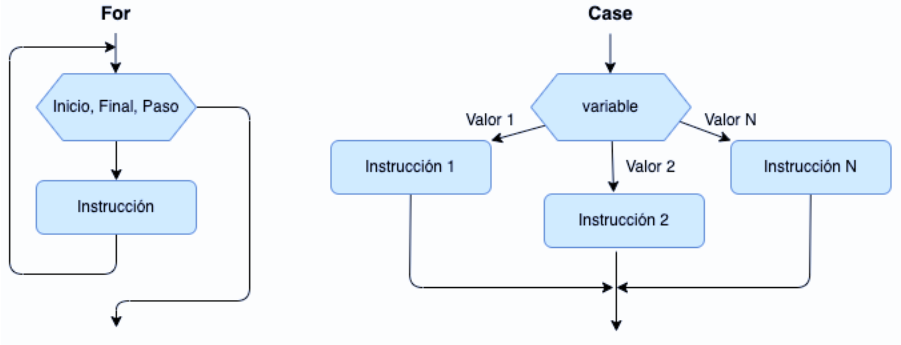
- Asignación
- Llamada a rutina(procedimiento o función)
- Bloque (secuencia de intrucciones)
- Estructuras de control
  - Condición (If, If-Else)
  - Repetición (For, While, Do-While, Repeat-Until)
  - Bifurcación (Case)

Cada una de las estructuras de control en sí misma una instrucción. La asignación como instrucción se representa como una sola instrucción y lo mismo ocurre para las llamadas a rutina. Para el caso de llamadas a rutina de Lectura de Datos y Escritura de Resultados tienen su propia símbología especial, la cual se ya se dio a conocer. Las siguientes figuras muestran los diagramas de flujo de cada una de la instrucciones de acuerdo a la clasificación.









Nótese que cada instrucción indicada puede ser la representación de otra instrucción y de esto, uno podría sustituir la instrucción de cada una de las estructuras de control (ya que sólo ejecuta una instrucción) o por un bloque lo cual llevaria a ejecutar una secuencia de instrucciones en vez de una sóla, o bien podría ser sustituida por otra estructura de control e inclusive por la misma estructura, a lo que se denómina *estructuras anidadas*.

Práctica 2.7. Elaborar el diagrama de flujo en VS code para el siguiente problema: Se desea calcular el pago a un trabajador si se registra el total de horas trabajadas y se sabe que el pago por hora normal es de 20 pesos y la hora extra a 30 pesos. También se sabe que como máximo 40 son las horas normales. Posteriormente el elabore el mismo diagrama en PSeInt y ejecutelo para determinar si esta correcto.

Ejercicio 2.4. Modificar el algoritmo de la práctica 2.7 de modo que incluya un descuento o dos descuentos (digamos impuesto y aportación a caja de ahorro). Los porcentajes usted los define.

Práctica 2.8. Elaborar el diagrama de flujo en VS code para el problema de leer tres números y determinar cúal es el menor y cúal es el mayor. Posteriormente el elabore el mismo diagrama en PSeInt y ejecutelo para determinar si esta correcto.

Práctica 2.9. Elaborar el diagrama de flujo en VS code para el problema de mostrar las tablas de las operaciones aritmeticas (suma, resta, multiplicación y división) para un número dado. Posteriormente el elabore el mismo diagrama en PSeInt y ejecutelo para determinar si esta correcto.

Práctica 2.10. Elaborar el diagrama de flujo en VS code para el problema de mostrar los números primos que existen entre el 1 y el 20. Posteriormente el elabore el mismo diagrama en PSeInt y ejecutelo para determinar si esta correcto.

Ejercicio 2.5. Elaborar el diagrama de flujo en VS code para el problema de encontrar un número (dado de manera predeterminada). En cada intento para encontrarlo se mostrarían los posibles mensajes: “Adivinaste en 4 intentos”, el 4 es como ejemplo, “Esta cerca”, Esta muy cerca”, “Esta lejos” y “Esta muy lejos”,

Ejercicio 2.6. Elaborar el diagrama de flujo en VS code para el problema de saber si se puede introducir un prisma rectangular por un orificio circular que esta en la pared. Usted defina las dimensiones de cada una de las figuras.

Ejercicio 2.7. Elaborar los diagrama de flujo en VS code para mostrar las tablas de verdad de los ejercicios 1.9 y 1.10 (antro).

Ejercicio 2.8. Elaborar algoritmo en diagrama de flujo de datos en VS code del ejemplo 2.1. Limite el algoritmo (diagrama) a un solo acceso, esto es, eliminar el proceso de la repeticion. Los mensajes de salida serían: “Permitir acceso” o “Negar acceso”.

Ejercicio 2.9. Modifique algoritmo del ejercicio 2.8 de modo que muestre uno de los siguientes mensajes:

- Acceso negado
- Permite acceso a dama.
- Permite acceso a dama y caballero.
- Permite acceso a menor.

Ejercicio 2.10. Modifique algoritmo del ejercicio 2.9 de modo que incluya la repetición de los accesos y que cuando termine se muestre el mensaje “Cerrando sistema. . . “

2.4 Representación de Algoritmos mediante Pseudocódigo

2.4.1 Uso de pseudocódigo para visualizar algoritmos

El propósito es familiarizarse con el uso del pseudocódigo como una herramienta efectiva para visualizar algoritmos. La habilidad para expresar algoritmos de manera clara y comprensible es esencial en el proceso de diseño de software.

Definición de Pseudocódigo

El pseudocódigo es un lenguaje intermedio entre el lenguaje natural y un lenguaje de programación real. Utiliza expresiones del lenguaje natural y código similar a un lenguaje de programación para definir los pasos que indica el algoritmo. El propósito real del pseudocódigo es comprender de manera clara el algoritmo y acercarse a la codificación a un lenguaje de programación.

Símbolos y Convenciones

El pseudocódigo introduce símbolos y convenciones de acuerdo al autor, al grupo de trabajo o a los estandares de la corporación. En este apartado se visualiza como se definen las variables, constantes, los tipo de datos, los operadores, la asignación variables, la formación de expresiones, las estructuras de control, la definición y llamada a rutinas y la documentación mediante comentarios.

Claridad y Simplicidad

Prestar atención en la importancia de mantener el pseudocódigo claro y fácil de entender.

Adaptabilidad a Diferentes Lenguajes

Se resalta que el pseudocódigo es independiente del lenguaje de programación. El pseudocódigo puede adaptarse fácilmente a diferentes lenguajes de programación. Esta característica hace que el pseudocódigo sea una herramienta versátil en el diseño de algoritmos.

Traducción a Código Real

Experimentar la traducción del pseudocódigo a código real en un lenguaje de programación específico. Esto implica a conocer las equivalencias de los símbolos y convenciones del pseudocódigo a las del lenguaje de programción en cuestion. Este paso se le denomina implementación del programa.

Utilizar el pseudocódigo como una herramienta de planificación y como comprensión de la estructura y la lógica del algoritmo antes de la implementación es de suma relevancia. Para muchos programadores de vasta experiencia podrían omitir el pseudocódigo y pasar

directamente a la implementación. Aunque como se menciono con anterioridad se deben elaborar para cuestiones de documentación.

2.4.2 Símbolos y convenciones del pseudocódigo

Como se menciono, dependiendo del autor o corporación se definen los símbolos y convenciones en pseudocódigo. Aquí mostraremos los más comunes utilizando el lenguaje HPLang. Estos elementos son esenciales para expresar algoritmos de manera clara y efectiva.

Práctica 2.11. Instalar la aplicación HP Compiler que está en la carpeta del curso como app-v4.8.6.zip el cual leerá instrucciones en Pseudo código en el lenguaje HPLang. Paa ello debe instalar previamente Java JDK version 21. .

Literales

Las literales son los valores como tales que se pueden utilizar tales como el 123, 12.3, '1', "123" y True. Los cuales son respectivamente un número entero, un número real, un carácter una cadena y un valor booleano.

Declaración de Variables

Las variables son los elementos del pseudocódigo para guardar las literales (valores), los cuales suelen cambiar cada vez que se vayan ejecutando las instrucciones (pasos) del pseudocódigo. Las variables deben declararse previamente antes de poderse usar, esto es, poderle poner o asignarle un valor (literal).

Tipo de datos

Los tipos de datos definen el tipo de valor (literal) que tienen las variables con las cuales se forman los operandos de las expresiones.

Operandos y expresiones

Cuando se dice que un operando es numérico es porque la variable es de tipo Integer o Real. Los operandos numéricos también pueden ser los números enteros o reales tal cual. Cuando el operando es booleano es porque la variable es de tipo Boolean o los valores True o False.

Los operandos se utilizan para formar expresiones y dependiendo del operador se tiene la expresión respectiva.

Asignación de Variables

Utilizar el símbolo <- para representar la asignación de un valor o el valor de una expresión a una variable.

Ejemplo 2.16. A <- 10

Expresiones Aritméticas

Utilizar los operadores aritméticos comunes como +, -, *, /, *Modulus* y *Div* para formar expresiones aritméticas. Nótese que para fomar expresiones aritméticas los operandos deben ser numéricos. El resultado de una expresión aritmética es un valor numérico. Una expresión aritmética puede estar formada por un sólo operando aritmético.

Ejemplo 2.17. x <- a + b * c **Modulus** d **Div** e - f

Expresiones Lógicas

Utilizar los operadores lógicos comunes como And, Or y Not para realizar expresiones lógicas. Nótese que para fomar expresiones lógicas los operandos deben ser booleanos. El resultado de una expresión lógica es un valor booleano. Una expresión lógica puede estar formada por un sólo operando lógico.

Ejemplo 2.18. X <- **Not** A **And** B **Or Not** C

Expresiones Relacionales

Utilizar los operadores relacionales comunes como >, >=, <, <=, == y <> para realizar expresiones relacionales. Nótese que para fomar expresiones relacionales sólo tiene dos operandos las cuales son expresiones aritméticas. El resultado de una expresión relacional es un valor booleano.

Ejemplo 2.19. X <- a + b <= c **Modulus** 10

Condiciones

Las condiciones son expresiones lógicas o expresiones relacionales cuyo resultado es util para tomar una determinación en las diferentes estructuras de control. Como ejemplo de ellas se visualizan en los ejemplos de las estructuras de control.

Estructura de control If

La estructura If tiene la sintaxis If Condición Then Instrucción. En esta estructura se ejecutará la Instrucción siempre y cuando la Condición sea verdadera. En el lenguaje cotidiano se lee “Si la condición se cumple entonces ejecuta la instrucción”.

Ejemplo 2.20. **If** a + b <= c Modulus 10 **Then** a <- 100

Estructura de control If-Else

La estructura If-Else tiene la sintaxis If Condición Then Instrucción1 Else Instrucción2. En esta estructura se ejecutará la Instrucción1 siempre y cuando la Condición sea verdadera de lo contrario se ejecutará la Instrucción2. En el lenguaje cotidiano se lee “Si la condición se cumple entonces ejecuta la instrucción1 y sino ejecuta la instrucción2”.

Ejemplo 2.21. **If** ok **Then** a <- 100 **Else** a <- 200

Estructura de control For

La estructura For tiene la sintaxis For var = valorInicial To valorFinal [Step valorPaso] Do Instrucción. En esta estructura se ejecutará la Instrucción tantas veces como haya desde el valorInicial hasta el valorFinal. La parte de Step valorPaso es opcional y suele omitirse si el valorPaso es 1. En cada ciclo el valor de la variable var se incrementará de acuerdo al valor de valorPaso. En el lenguaje cotidiano se lee “Desde el valor inicial hasta el valor final ejecuta la instrucción” se puede agregar “incrementado en valorPaso”.

Ejemplo 2.22. **For** a <- 10 **To** 100 **Step** 3 **Do** b <- b + a

Estructura de control While

La estructura While tiene la sintaxis While Condición Do Instrucción. En esta estructura se ejecutará la Instrucción cada vez que la Condición sea verdadera. En el lenguaje cotidiano se lee “Mientras la condición se cumpla ejecuta la instrucción”.

Ejemplo 2.23. **While** a < b * 2 **Do** a <- a * 2

Estructura de control Do – While

La estructura Do - While tiene la sintaxis Do Instrucción While Condición. En esta estructura se ejecutará la Instrucción cada vez que la Condición sea verdadera. Esta estrutura es similar a While. La estructura Do-While se ejecuta la instrucción por lo menos una vez mientra que la estructura While es posible que no se ejecute la instrucción ni una sola vez. En el lenguaje cotidiano se lee “Ejecuta la instrucción mientras la condición se cumpla”.

Ejemplo 2.24. **Do** a <- a * 2 **While** a < b * 2

Estructura de control Case

La estructura Case tiene la sintaxis Case var Of valor1 : Instrucción1; valorN: instrucciónN [; Otherwise: InstrucciónOtherwise] End. En esta estructura se ejecutará la InstrucciónN si el valorN es igual al valor que tenga la variable var. En el caso de que ningún valorN coincida con el valor de var se ejecutaria la instrucciónOtherwise si es que estuviese presente dado a que es opcional que esté. En el lenguaje cotidiano se lee “En el caso de

que la variable valga valorN se ejecuta la instrucciónN" y si tiene la parte opcional Otherwisee, se agregaría a la leyenda "de lo contrario se ejecuta la instruccionOtherwise".

Ejemplo 2.25. **Case** a **Of** 10: a <- b * 2; 20: a <- b *3; **Otherwise**: a<- b*b **End**

Llamada al procedimiento Write() y Writeln()

Las llamadas al procedimiento Write() y Writeln() son utilizadas para escribir o mostrar mensajes o resultados al usuario (consola o terminal, ambos conceptos son similares, la diferencia esta bajo los conceptos del sistema operativo. Ya sea consola o terminal, se consideran dos dispositivos: el de entrada compuesto por el teclado y el de salida compuesto por el monitor. Se da por hecho que estos dispositivos los maneja el usuario y de ahí que cuando se dice "se leen datos del usuario" se entiende que entran por teclado y cuando "se muestran los resultados al usuario" los resultados se muestran en el monitor.) Los mensaje se encierran con comillas dobles y los resultados pueden ser variables o expresiones. La diferencia entre ambas llamadas es que Write() una vez de que termine de mostrar los mensajes o resultados el cursor se mantiene en la misma línea de la consola o terminal y Writeln() pone el cursor en la siguiente línea. Un Writeln() sin argumentos (mensajes o resultados) indicaría hacer un salto de línea.

Ejemplo 2.26. **Write**("Ingresa un número?")

Llamada al procedimiento Read()

La llamada al procedimiento Read() es utilizada para leer valores desde el teclado (usuario o consola o terminal). Por el momento sólo se lee una variable.

Ejemplo 2.27. **Read**(a)

Bloque Begin – End o bien { - }

Un bloque es una instrucción compuesta por varias instrucciones separadas cada una de ellas por un punto y coma.

Ejemplo 2.28. **Begin Write**("Ingresa un número?"); **Read**(a); **If** a > 100 **Then** a <- a -100 **End**

Comentarios

Los comentarios son de dos tipos: múltiples línea o de bloque y de una sóla línea. No son ejecutables y sólo son para documentar el pseudocódigo. Se utiliza /* para iniciar el comentario de bloque y termina con */. Se utiliza // para iniciar el comentario de una línea y termina hasta el final de la línea. Ambos tipos de comentarios se pueden colocar en cualquier parte del pseudocódigo.

Ejemplo 2.29. Read(a); **/*** Esta es una sesión en HPLang ***/** If a > 100 // Falta Then.

Práctica 2.10. Elaborar los pseudocódigos en HPLang para las prácticas de la sección 2.3 correspondientes a los diagramas de flujo solicitados.

Ejercicio 2.11. Elabore los pseudocódigos en HPLang para los ejercicios de la sección 2.3 correspondientes a los diagramas de flujo solicitados.