1. Caso práctico.

Unidad Didáctica IV

Caso práctico

SI Andalucía SCA *es una empresa que busca soluciones, y si no las encuentra las diseña y las construye. En el mundo de la informática hacer una aplicación es una tarea apasionante, porque entre otras cosas, antes de diseñar la solución es imprescindible conocer a fondo el problema y cuales son las necesidades del cliente.*

Es muy diferente trabajar en el sector informático como ***experto en desarrollo*** *de software (aplicaciones de escritorio o de Internet) o como* ***instalador de infraestructuras*** *de sistemas informáticos (redes locales u oficinas con ordenadores), porque aunque en ambos puestos se requiere hacer uso de todos los conocimientos adquiridos, de técnicas contrastadas y de las herramientas y recursos adecuados, en el ámbito de la programación se precisa además de cierta dosis de creatividad que hace que un problema tenga diferentes soluciones y diferentes programas que la sinteticen, y en cada uno de estos programas podemos encontrar preferencias de unas personas con respecto a otras.*

La programación de ordenadores debe basarse en un riguroso estudio previo que finaliza en la confección de unos algoritmos, de los que va a depender todo el posterior proceso de desarrollo de una aplicación, y el éxito de la misma en la mayoría de los casos depende de la correcta elaboración de estos algoritmos.

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

2. Concepto y definición de algoritmo.

Unidad Didáctica IV

Concepto y definición de algoritmo

*Todos los programadores recuerdan sus inicios en el diseño y elaboración de algoritmos y **Víctor** no es una excepción. Le viene a la mente cuando en la empresa de sus amigos le enseñaron a diseñar algoritmos porque, en aquel momento, era necesario que todos los miembros del equipo de programación asumieran una parte concreta de la aplicación y según ellos, el tiempo que "perdían" en enseñarle lo daban por bien empleado ya que lo recuperarían rápidamente durante la fase de implementación de la aplicación.*

*A **Víctor** le hizo sentir bien aquella confianza que depositaban en él y pensaba que no necesitaba aprender a elaborar algoritmos porque ya sabía programar, así que no les fallaría. Los problemas vinieron después cuando el equipo de trabajo le obligaba a seguir un método de trabajo necesario para el desarrollo de una aplicación en equipo. Aquello no le gustó, pero de inmediato vio las ventajas de las contrastadas técnicas de programación y lo importante que era disponer de un algoritmo para cada una de las tareas a programar.*

Como hemos comentado en las unidades anteriores, con la programación intentamos solucionar problemas usando los ordenadores, que ante todo lo que nos aportan es rapidez. ¿Qué será lo primero que tendremos que hacer para solucionar cualquier problema? Parece que encontrar una forma de solucionarlo, una solución. Y una vez que dispongamos de esa solución, ¿cómo hacer que sea comprensible por un ordenador?

La respuesta a esas preguntas son los **algoritmos.**

La popularización del término algoritmo ha llegado asociada a la era informática, pero el término algoritmo proviene de **Mohammed al-Khowarizmi**, matemático persa que vivió durante el siglo IX y alcanzó gran reputación por el enunciado de las reglas para sumar, restar, multiplicar y dividir números decimales. La traducción al latín del apellido como la palabra algorismus derivó posteriormente en algoritmo. **Euclides**, matemático griego (del siglo IV antes de Cristo) que inventó un método para encontrar el máximo común divisor de dos números, se considera con **Al-Khowarizmi** el otro gran padre de la **algoritmia** (ciencia que trata de los algoritmos).

Niklaus Wirth, inventor entre otros de los lenguajes de programación Pascal y Modula-2, tituló uno de sus más famosos libros**, Algoritmos + Estructuras de Datos = Programas**, destacando el hecho de que sólo **se puede llegar a realizar un buen programa con el diseño de un buen algoritmo y una correcta utilización de las estructura de datos**. La resolución de un problema exige el diseño de un algoritmo que resuelva el problema propuesto.

Un **algoritmo** es:

- **Una "fórmula" para resolver un problema.**
- **Un conjunto finito de acciones o secuencia de operaciones que ejecutadas en un determinado orden resuelven el problema.**
- **También puede definirse como un método para resolver un problema mediante una serie finita de pasos precisos y bien definidos**.

El **lenguaje algorítmico** es una forma de representar los algoritmos de manera adecuada para que la solución sea clara para cualquier persona.

El conjunto de todas las operaciones a realizar junto al orden en que deben efectuarse, se denomina algoritmo, y el lenguaje que permite representar esa solución de forma clara, y sin ambigüedades, es el lenguaje algorítmico.

El **programador** de computadoras es antes que nada una persona que resuelve problemas, por lo que para llegar a ser un programador eficaz se necesita aprender a resolver problemas de un modo riguroso y sistemático. A la metodología necesaria para resolver problemas mediante programas se denomina **Metodología de la Programación**. El eje central de esta metodología es el concepto de algoritmo.

Ejemplos usuales de algoritmos son una receta de cocina, o el protocolo de actuación de un médico para atender y curar a un paciente que padece una determinada enfermedad.

En el contexto de la informática, el algoritmo representa la secuencia de acciones o instrucciones que debe ejecutar el ordenador para solucionar un problema.

Víctor *recuerda una ocasión en la que* ***Carmen*** *explicó el concepto de algoritmo a un amigo que se interesó por su función en la empresa. Y la mejor forma de explicárselo fue con una receta de cocina, concretamente con el último postre que hizo cuando invitó a comer a los compañeros de la empresa. Se trataba de una sencilla tarta de manzana con una elaboración muy simple y con la que quedó muy bien ante sus invitados. Sus argumentos fueron que la elaboración de un plato de cocina requiere una planificación previa y seguir unos pasos de forma ordenada para conseguir un resultado (en este caso el postre).*
Si te interesa la cocina y quieres conocer el algoritmo para elaborar una deliciosa tarta de manzana (comúnmente llamada tarta de polvos) visualiza la siguiente presentación:

Demo: Es muy importante conocer bien los algoritmos y empezar de forma adecuada, ya que toda tu posterior trayectoria como programador se va a basar en dominar esta técnica de solución de problemas.

AUTOEVALUACION

Un algoritmo es: (marca la opción correcta)

a) Un programa para resolver problemas
b) Un método para resolver un problema mediante una serie de datos precisos y definidos
c) Un método para resolver un problema mediante una serie de datos precisos, definidos y finitos
d) Un método para resolver un problema mediante una serie de datos precisos, definidos y concretos

Comprobar

¿Cuál de las siguientes características de un algoritmo es incorrecta?

a) Un algoritmo debe ser preciso e indicar el orden de realización de cada paso.
b) Si se sigue un algoritmo dos o más veces con los mismos datos,

- se pueden obtener distintos resultados cada vez.
- c) Un algoritmo debe ser finito. Si se sigue un algoritmo se debe terminar en algún momento; o sea, debe tener un numero finito de pasos
- d) Los algoritmos deben ser independientes del lenguaje de programación que se use y del ordenador en que se ejecuten.

Comprobar

PARA SABER MÁS:
En el siguiente enlace podrás profundizar en la importancia de saber resolver problemas y aprender a razonar.
Aprender a resolver problemas razonando. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web)
Para descargar el programa Acrobat Reader pulsa aquí.

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

3. Características de un algoritmo

Unidad Didáctica IV

Características de un algoritmo

Víctor *siempre ha pensado que un programa es bueno si funciona bien. Pero* ***José*** *no está de acuerdo con él y le hace ver lo equivocado que está.* ***José*** *explica que para que un programa sea bueno éste debe pasar por una serie de pruebas exhaustivas y de un tiempo de utilización de forma real. Y ante la sorpresa de* ***Víctor****, comenta que no todos los programas superan esas pruebas y que hay muchos que deben ser continuamente revisados, parcheados e incluso sustituidos por versiones más nuevas que en ocasiones son programas totalmente diferentes. También comenta que esto que comenzó siendo una práctica de incompetentes ahora se ha convertido en una técnica de mercado que está funcionando, y muy bien. Parece que ahora interesa sacar a la venta versiones incompletas que se van mejorando cada cierto tiempo para "obligar" al cliente a actualizarse o a adquirir una versión mucho más "potente", pero que también será incompleta.*

José *dice que no hay que confundir esto con la necesaria e inevitable evolución del software, pero que hay situaciones en las que esas mejoras no dejan de ser un mero "lavado de cara" con aportaciones prescindibles y con el objetivo prefijado de lanzar versión en breve. Añade que hubo un tiempo (no hace mucho) en el que los programas que se desarrollaban cumplían todas las características de lo que debe ser un buen programa y que de hecho algunos de ellos aún no han sido sustituidos, es el caso del software empleado por los controladores aéreos de algunos aeropuertos que mantienen los programas de hace treinta años y actualmente son incapaces de mejorarlos.*

Cualquier algoritmo no será válido si lo que pretendemos es usarlo para solucionar un problema por medio de ordenadores. Si **José** sabe resolver un problema, basta con que le explique cómo resolverlo a **Víctor** para que éste también sepa resolverlo. Pero hay una diferencia, es posible que **José** sepa además resolverlo mediante un programa, mientras que es posible que **Víctor** no sepa.

No es lo mismo saber resolver un problema que saber plantearlo de forma clara para que una máquina (ordenador) lo solucione.

Para ser útil, y sobre todo para ser programable, el algoritmo debe cumplir una serie de características, que faciliten su trascripción a un lenguaje de programación y que garanticen que la solución va a ser la más eficaz posible.

- Un algoritmo debe ser **preciso, sin ambigüedades**, **e indicar el orden** de realización de cada paso.
- Un algoritmo debe estar **bien definido**. Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.
- Un algoritmo debe ser **finito**. Si se sigue un algoritmo se debe terminar en algún momento; o sea, debe tener un número finito de pasos.
- Un algoritmo es **independiente tanto del lenguaje de programación en que se codifica como de la computadora que lo ejecuta**. Cada programador puede expresar el algoritmo en un lenguaje diferente de programación y ejecutarlo en una computadora distinta. Sin embargo, el algoritmo será siempre el mismo. Así, por ejemplo, en una analogía con la vida diaria, una receta de un plato de cocina se puede expresar en español, inglés o francés, pero cualquiera que sea el lenguaje, los pasos para la elaboración del plato se realizarán sin importar el idioma del cocinero.
- **En programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras**. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo. Tanto el lenguaje de programación como la computadora son los medios para obtener un fin: conseguir que el algoritmo se ejecute y se efectúe el proceso correspondiente. El diseño de la mayoría de los algoritmos requiere creatividad y conocimientos profundos de la técnica de la

programación.

- **En la mayoría de los casos existen varios algoritmos para la solución de un problema dado**. Hay que coger el más eficiente, es decir, el que resuelva el problema más rápidamente y usando la menor cantidad de recursos posibles del ordenador (tiempo de CPU y memoria, fundamentalmente)
- La definición de un algoritmo suele incluir tres partes: Entrada, Proceso y Salida. En el algoritmo de la receta de cocina citado anteriormente se tendrá:

Entrada: ingredientes y utensilios empleados.
Proceso: elaboración de la receta en la cocina.
Salida: presentación del plato (por ejemplo, cordero).

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

4. Clasificación de problemas.

Unidad Didáctica IV

Clasificación de problemas

Víctor *le pregunta a* ***José*** *si él, que ya tiene experiencia en el diseño de algoritmos, sería capaz de desarrollar un algoritmo para solucionar cualquier problema. Pero la respuesta de José le deja un poco "chafado". Dice que no todos los problemas tienen solución y que a veces tras un estudio de viabilidad debemos decirle al cliente que el problema que tiene no es posible solucionarlo con el uso de ordenadores y que esto sucede en muchas ocasiones. Para poner un ejemplo le comenta que existen programas para jugar al ajedrez que son muy buenos, incluso podemos pensar que existe un algoritmo para ganar siempre a este juego, pero éste es uno de esos problemas que ahora no puede ser resuelto, ya que el número de operaciones, comparaciones y situaciones a tener en cuenta lo hacen intratable para la actual tecnología informática.*

Justamente porque no es lo mismo entender un problema que resolverlo, nos podemos encontrar con problemas de muchos tipos. ¿Piensas que todos los problemas tienen solución? ¿Piensas que si sabemos que un problema tiene solución, siempre sabremos encontrarla? ¿Te has encontrado alguna vez con algún problema que sabes cómo se debería resolver, pero implica tal cantidad de trabajo y tiempo que no es posible plantearse su solución? Seguramente sí. Hay tareas que son hasta sencillas, pero que requieren tal cantidad de pasos que no podemos pensar siquiera en darlos todos en varias generaciones.

Por eso los problemas se pueden dividir en primera instancia en dos grupos:

- **Problemas indecidibles**: aquellos que no se pueden resolver mediante un algoritmo.
- **Problemas decidibles**: aquellos que cuentan al menos con un algoritmo para su cómputo.

Sin embargo, que un problema sea decidible no implica que se pueda encontrar su solución. Muchos problemas que disponen de algoritmos para su resolución son inabordables para un ordenador por el elevado número de operaciones que hay que realizar para resolverlos. Esto permite separar los problemas **decidibles** en dos:

- **Intratables**: aquellos para los que no es factible obtener su solución.
- **Tratables**: aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable.

AUTOEVALUACIÓN

Indica cuál de las siguientes afirmaciones es correcta:

- a) Problemas indecidibles son aquellos para los que aunque exista una solución es imposible calcularla.
- b) Problemas decidibles son aquellos que cuentan con más de un algoritmo para su cómputo
- c) Problemas indecidibles o intratables son aquellos para los que no es factible obtener su solución
- d) Problemas tratables son aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable

Comprobar

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

Unidad Didáctica IV

Ejemplos de la necesidad de diseñar algoritmos eficientes

Igual que ocurre en el ajedrez, hay muchas situaciones en las que es necesario hacer un buen estudio del problema a fin de encontrar su solución. José le explica que en una ocasión, cuando era estudiante, en una de las clases se obtuvieron cuatro algoritmos diferentes para resolver un mismo problema y que resultó muy interesante porque después su profesora analizó los algoritmos para establecer cuál presentaba mayor eficacia en rapidez de ejecución, uso de la memoria, accesos a disco, etc. Pero que en aquella ocasión comprendió que aunque todos los algoritmos eran una solución óptima al problema, sólo uno de ellos era el más adecuado y eficiente para su implementación.

Piensa que nuestro problema es tan simple como realizar la multiplicación de 25 * 100.

Incluso en este caso, puedes encontrar distintas formas de resolverlo, distintos algoritmos de multiplicación:

- Sumar 25 consigo mismo 100 veces
- Sumar 100 consigo mismo 25 veces.
- Añadir dos ceros a 25, ya que estamos multiplicando por una potencia de 10, que es la base de numeración.
- Hacer la multiplicación típica.

Evidentemente, no todas las formas de resolverlo son igual de buenas, ni igual de eficientes, no se tarda lo mismo en hacerlas, ni son igual de fáciles.

Lo mismo ocurre si pensamos en casos reales en los que elegir algoritmos adecuadamente marca la diferencia entre que el resultado sea útil o no. En los siguientes apartados tratamos de que **percibas lo importante que puede llegar a ser diseñar un algoritmo adecuado, y que intuyas lo complicado que puede llegar a ser en algunos casos**. No obstante, los algoritmos con los que empezaremos a hacer prácticas son bastante más elementales, y se irá introduciendo la complejidad de forma gradual a lo largo de todo el módulo profesional.

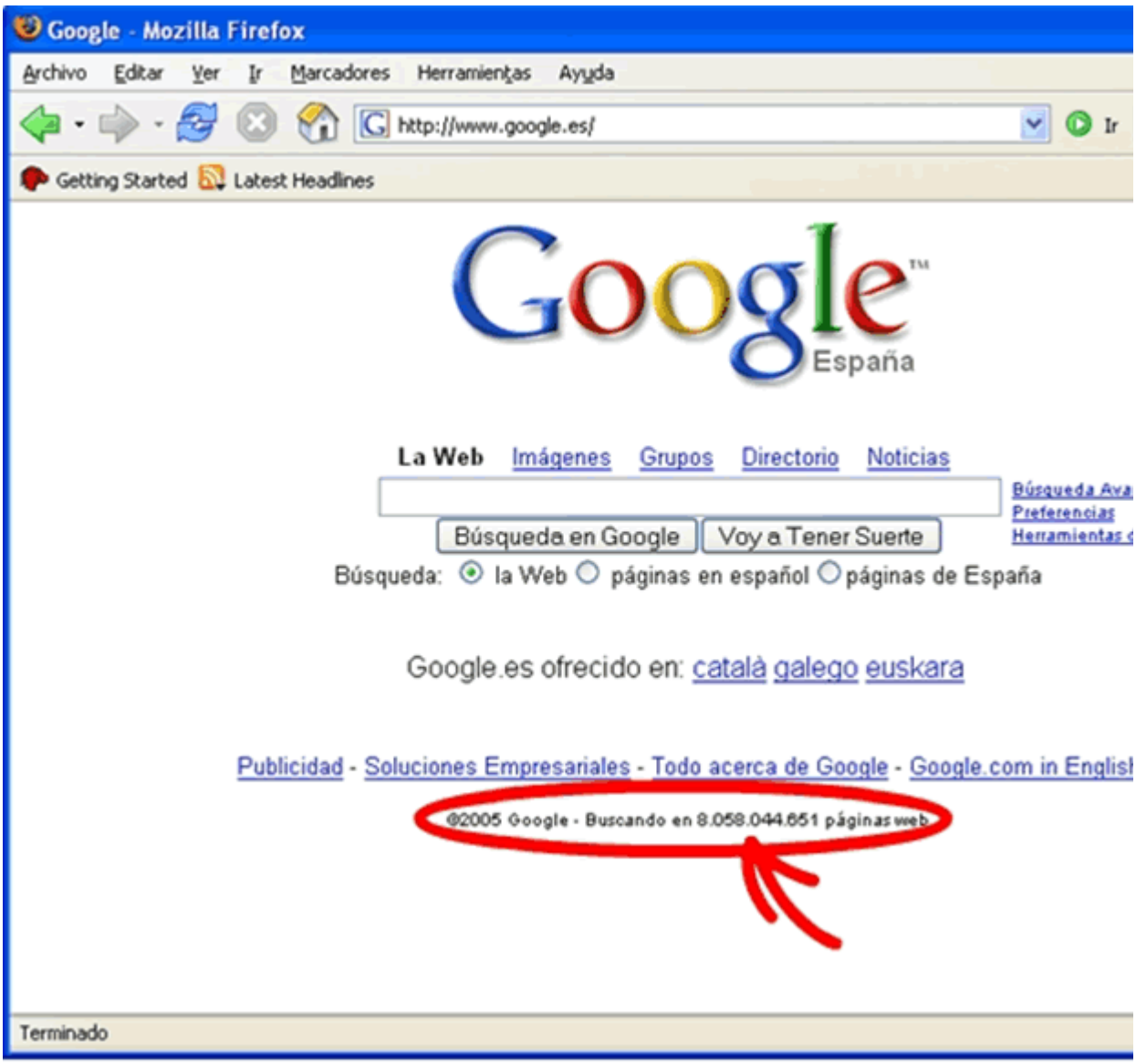
5.1. Motor de búsquedas

Unidad Didáctica IV

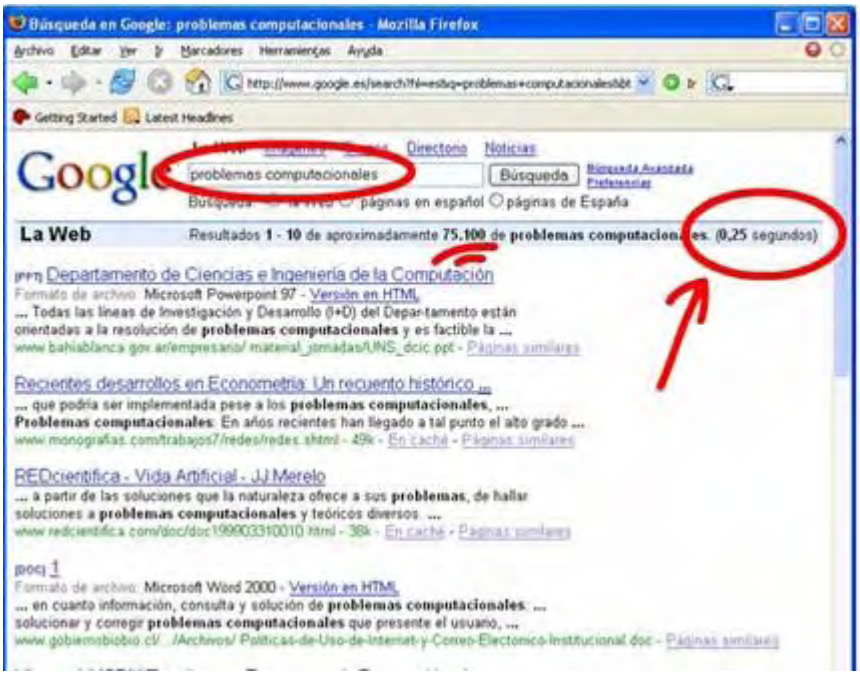
Motor de búsquedas

Seguramente habrás utilizadoInternet en alguna ocasión, o habrás oído a alguien hablar de que ha buscado alguna cosa en Internet. Incluso puede que te haya dicho que lo ha buscado con Google, que es un popular buscador, o **motor de búsqueda.**

Entendemos por **motor de búsqueda** (o sencillamente **buscador**) una **aplicación** de **Internet** que permite localizar direcciones cuyos contenidos se relacionan con el tema buscado. Evidentemente, para buscar y seleccionar direcciones útiles, que tengan alguna relación con el tema buscado de entre los millones disponibles en todo Internet, el algoritmo debe ser muy eficiente.



...Buscando en 8.058.044.651 páginas Web.



Buscador de Internet: ¡¡¡ Más **de Ocho mil millones de páginas** revisadas **en 25 centésimas** de segundo y **75.100 encontradas y clasificadas** por su grado de similitud o interés!!!

Problema:

- ¿Cómo estructurar la información necesaria para realizar las consultas rápidamente?
- ¿Qué algoritmos de búsqueda utilizar?

PARA SABER MÁS:
Si quieres ver otro motor de búsqueda, en este caso de recuperación de la información para uso experimental y educativo, visita el siguiente enlace:
Recuperación de la información en la educación. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web).

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

5.2. Planificador de rutas

Unidad Didáctica IV

Planificador de rutas

¿Quién no se ha ido de viaje alguna vez y ha tenido que tirar del mapa para ver qué camino es el más aconsejable? Y no sólo eso. ¿Piensas que cuando un repartidor hace una ruta, o cuando los camiones de basura eligen un recorrido o cuando los carteros reparten las cartas en el barrio, siguen un camino al azar? Pues claro que no.

Se trata de **encontrar el camino más corto entre dos puntos, o el camino más corto que pase por todos los puntos intermedios**. Es un problema con múltiples aplicaciones en la vida cotidiana para elaborar las mejores rutas (más cortas y más rápidas) para carteros, repartidores, rutas de autobuses, dispositivos GPS de los vehículos, etc.

- ¿Cómo representar la información (lugares y carreteras)?
- ¿Cómo calcular el camino más corto entre dos lugares?
  - Representación mediante un grafo:
  - Lugares = nodos.
  - Carreteras = arcos entre nodos.

¿Cómo calcular los caminos mínimos en el mapa?

- Fuerza bruta: empezar por Cagitan, que es el origen (ver imagen) y probar todos los caminos hasta llegar a La Coruña.
- Supongamos que limitamos el problema a un mapa de 20 ciudades, existiendo 6 caminos por ciudad. ¡¡Existen 95 billones de caminos!!

AUTOEVALUACIÓN

Referido a algoritmos, señala la afirmación correcta:

a) La definición de un algoritmo suele incluir tres partes: Entrada, Proceso y Salida.

b) En programación, los algoritmos son más importantes que los lenguajes de programación o las computadoras. Un lenguaje de programación es tan sólo un medio para expresar un algoritmo y una computadora es sólo un procesador para ejecutarlo

c) En la mayoría de los casos existen varios algoritmos para la solución de un problema dado. Hay que coger el más eficiente, es decir, el que resuelva el problema más rápidamente y usando la menor cantidad de tiempo de CPU y de memoria.

d) Todas las anteriores son correctas

Comprobar

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

Unidad Didáctica IV

Diseño y representación de algoritmos

En una empresa como SI Andalucía formada por jóvenes titulados como Técnicos Superiores en Desarrollo de Aplicaciones Informáticas, es normal que cada uno de ellos tenga sus preferencias a la hora de representar los algoritmos que están desarrollando.

En el momento que vive ahora esta empresa, están desarrollando una aplicación con un tamaño importante y requieren que todos "arrimen el hombro" y se pongan a diseñar algoritmos y por supuesto, no hay inconveniente en que cada uno de ellos utilice el método de presentación del algoritmo que prefiera.

Pero para Víctor, que está empezando a abordar un problema y aprendiendo a hacer algoritmos esto es un quebradero de cabeza, porque intenta comparar los diseños de sus compañeros y no ve nada en común. Por eso cuando María le dedica unos días a enseñarle todos los modos de representar un algoritmo y a explicarle las ventajas que cada uno de sus compañeros aprecia en cada uno de ellos, entiende que no son tan diferentes y que cada uno se sienta más cómodo con su preferido. María le dice que en una semana dedicado a diseñar algoritmos él también tendrá su preferido y será con el que elabore todos los algoritmos en adelante, aunque será capaz de interpretar cualquier algoritmo independientemente del modo en que su autor lo haya representado, incluso aunque no sea ninguno de estos.

Hemos fijado el concepto de algoritmo, y hemos establecido la necesidad de encontrar algoritmos adecuados y eficientes. ¿Pero cómo representamos los algoritmos? ¿Es necesario seguir algunas normas? ¿Hay algún convenio para describir la solución a un problema? ¿Debemos preocuparnos del lenguaje de programación que luego usaremos?

Una vez comprendido el problema se trata de determinar qué pasos o acciones tenemos que realizar para resolverlo.

Como criterios a seguir a la hora de dar la solución algorítmica hay que tener en cuenta:

1. **Si el problema es bastante complicado lo mejor es dividirlo en partes más pequeñas e intentar resolverlas por separado. Esta metodología de "divide y vencerás" también se conoce con el nombre de diseño descendente.**
2. **Las ventajas de aplicar el diseño descendente son:**
   - **Al dividir el problema en módulos o partes se comprende más fácilmente.**
   - **Al hacer modificaciones es más fácil realizarlas sobre un módulo en particular que en todo el algoritmo.**
   - **En cuanto a los resultados, se probarán mucho mejor comprobando si cada módulo da el resultado correcto que si intentamos probar de un golpe todo el problema porque si se produce un error sabemos que módulo ha sido el responsable.**
3. **Una segunda filosofía a la hora de diseñar algoritmos es el refinamiento por pasos,** que consiste en partir de una idea general e ir concretando cada vez más esa descripción hasta que tengamos algo tan concreto para resolver que no represente ninguna dificultad. Pasamos de lo más complejo a lo más simple.

Una vez que tenemos la solución hay que implementarla con alguna representación. Las **representaciones** más usadas son los **flujogramas** (o diagramas de flujo), los **diagramas NS** y el **pseudocódigo**.

También la solución se podría escribir en algunos casos en lenguaje natural pero no se hace porque éste es muy ambiguo.

Al escribir el algoritmo hay que tener en cuenta:
Las acciones o pasos a realizar han de tener un determinado orden.

- **En cada momento sólo se puede ejecutar una acción o sentencia (también llamada instrucción).**
- **Dentro de las sentencias del algoritmo pueden existir "palabras reservadas", es decir, palabras a las que se les da un determinado significado, y que no podremos usar para nada distinto.**
- **Si estamos utilizando pseudocódigo tenemos también que usar la indentación, ya que aumenta la legibilidad del problema para que se pueda entender mejor.**

DEMO: En esta simulación puedes encontrar tres representaciones diferentes del mismo algoritmo y comparar cada uno de los pasos del algoritmo en cada una de las representaciones.

AUTOEVALUACIÓN:

Al escribir el algoritmo hay que tener en cuenta

a) En cada momento sólo se puede ejecutar una acción o sentencia o instrucción
b) Dentro de las sentencias del algoritmo pueden existir "palabras reservadas", es decir, palabras a las que se les da un determinado significado, y que no podremos usar para nada distinto.
c) Si estamos utilizando pseudocódigo tenemos también que usar la indentación, ya que aumenta la legibilidad del problema para que se pueda entender mejor.
d) Todas las anteriores son correctas

Comprobar

Al escribir el algoritmo hay que tener en cuenta

- a) En cada momento sólo se puede ejecutar una acción o sentencia o instrucción
- b) Dentro de las sentencias del algoritmo pueden existir "palabras reservadas", es decir, palabras a las que se les da un determinado significado, y que no podremos usar para nada distinto.
- c) Si estamos utilizando pseudocódigo tenemos también que usar la indentación, ya que aumenta la legibilidad del problema para que se pueda entender mejor.
- d) Todas las anteriores son correctas

Comprobar

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

6.1. Diagramas de flujo o flujogramas.

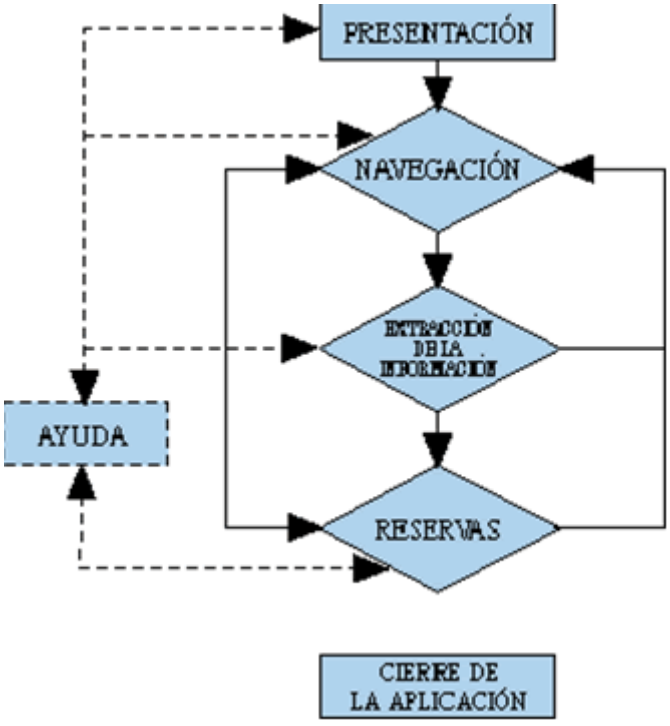
Unidad Didáctica IV

Diagramas de flujo o flujogramas

En el apartado anterior hemos mencionado tres de las formas usuales de representar los algoritmos. La primera de ellas son los diagramas de flujo.

No son más que una herramienta gráfica, una notación gráfica estándar que posibilita la comprensión de los pasos y el orden en que hay que darlos para resolver un problema (algoritmo).

Se basa en la utilización de unos símbolos gráficos que denominamos cajas, en las que escribimos las acciones que tiene que realizar el algoritmo. Deben ser leídos de arriba abajo, y de izquierda a derecha. Tienen la ventaja de proporcionar una representación muy visual, que permite entender de un solo golpe de vista la estructura del algoritmo, el tipo de acciones que se realizan y el orden en que se realizan. Son muy adecuados para algoritmos que no requieren diagramas muy grandes, pero se hacen difíciles de manejar cuando un solo diagrama ocupa muchas páginas. Además, resultan engorrosos de corregir cuando se detectan errores.



Las cajas están conectadas entre sí por líneas y eso nos indica el orden en el que tenemos que ejecutar las acciones. En todo algoritmo siempre habrá una caja de inicio y otra de fin, para el principio y final del algoritmo.

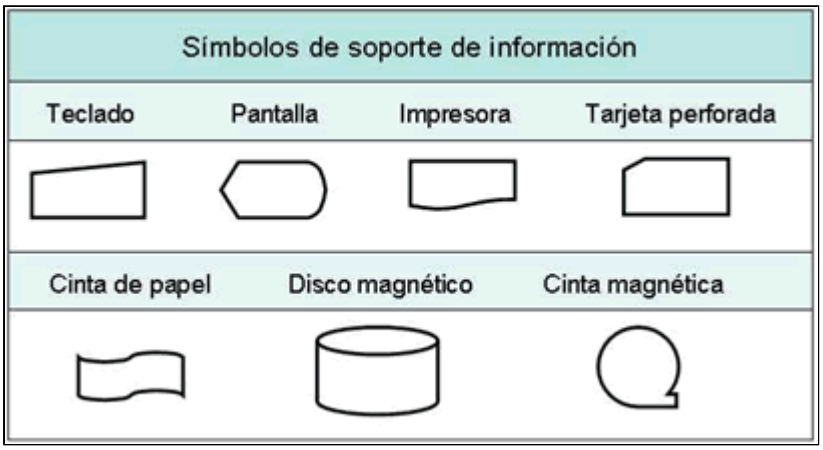
6.1.1. Símbolos usados en los diagramas de flujo.

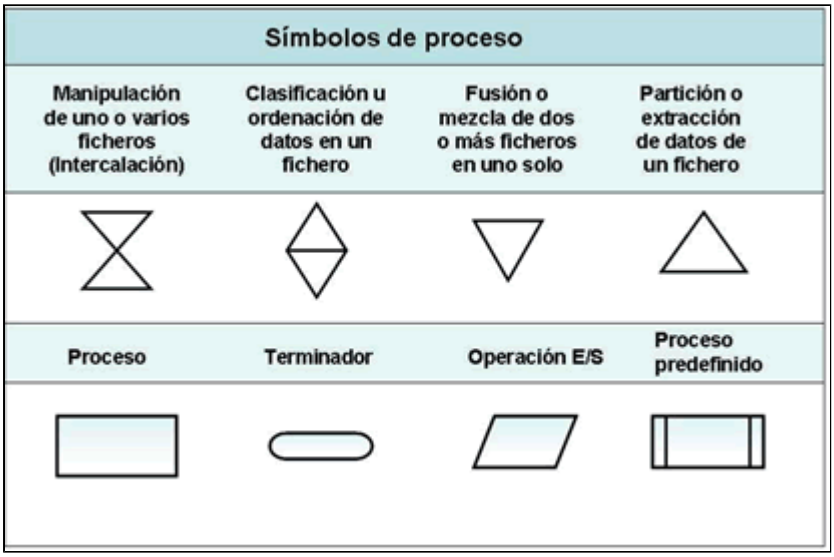
Unidad Didáctica IV

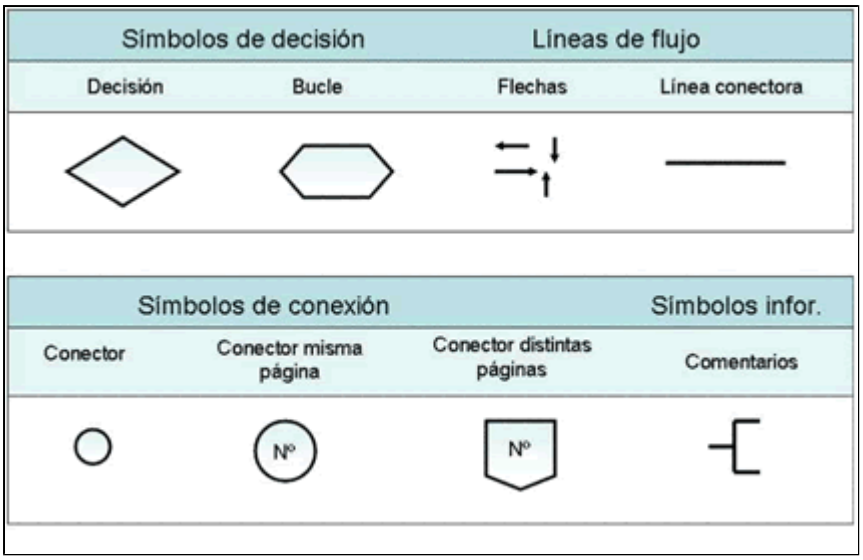
Símbolos usados en los diagramas de flujo

Aquí tienes una aplicación que de forma interactiva va a permitirle consultar rápidamente cada uno de los símbolos utilizados en los diagramas de flujo.

A continuación te presentamos un cuadro de la clasificación de los símbolos utilizados en los diagramas de flujo. Son los mismos que aparecen en el ejemplo anterior, pero hemos pensado que te puede ser útil que además los tengas todos aquí más a mano, para que utilices el sistema que mejor se adapte a tu forma de estudiar.







Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

6.1.2. Problema ejemplo

Unidad Didáctica IV

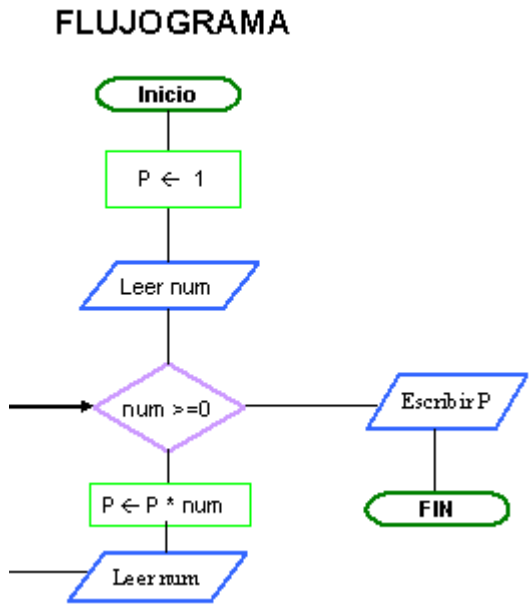
Problema ejemplo

En los apartados anteriores hemos comentado las características de los diagramas de flujo, y los símbolos que usan. Es el momento de que veamos un ejemplo.

Queremos **hallar el producto de varios números positivos introducidos por teclado y el proceso termina cuando se introduce un número negativo**. Los pasos necesarios son los siguientes:

1. Inicializar la variable del producto (asegurarnos de que vale 1).
2. Leer el primer número.
3. Preguntar si el número leído es positivo.
4. Si el número leído es positivo, multiplicamos el producto acumulado por el número leído, luego leemos un nuevo número, y se vuelve al paso 3.
5. Si por el contrario el número leído es negativo, escribimos el producto acumulado y terminamos.

El diagrama de flujo que representa esa solución es el siguiente:



DEMO: Puede ver un ejemplo de la ejecución de este flujograma en la siguiente simulación. A cada clic de ratón se irá ejecutando el algoritmo, de forma que puedes comparar la ejecución de cada paso en dos representaciones diferentes (flujograma y pseudocódigo). Deberías intentar anticiparte a cada paso antes de avanzar al siguiente.

DEMO: Otro ejemplo del mismo algoritmo, nos va a demostrar una ejecución diferente. Puedes verlo en esta otra animación

PARA SABER MÁS:
Enlace en el que podamos descargar alguna aplicación gratuita para hacer diagramas de flujo. Puedes encontrar varias opciones para Windows.
Programas de para elaborar Diagramas de Flujo.
Para entornos Linux tienes otras opciones, entre las que destacamos el enlace a DIA, un generador de diagramas de flujo muy útil. En esta página puedes encontrar una breve descripción y algunos enlaces interesantes:

Diagramas de Flujo en Linux. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web).

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

6.2. Diagramas N-S o de Nassi-Shneiderman:

Unidad Didáctica IV

Diagramas N-S o de Nassi-Shneiderman:

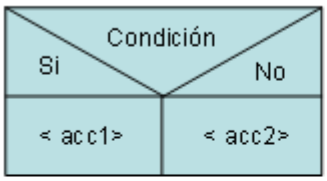
Los diagramas N-S son la segunda forma de representar los algoritmos. Son semejantes a los flujogramas, en el sentido de que también son una representación gráfica, pero sin flechas y cambiando algo los símbolos de condición y repetición.

Las cajas van unidas. Pretenden presentar las acciones de forma más parecida a como se introducirán en el programa, al igual que se hace en pseudocódigo, pero sin perder la ventaja visual (claridad de un simple vistazo) de los diagramas de flujo. La verdad es que lo consiguen a medias, y sólo para el caso de pequeños algoritmos, en los que no se necesita más de una página, y en los que no hay bloques grandes de sentencias que representar. Es por eso que se usan poco, y nos vamos a ocupar poco de su estudio, aunque debemos conocerlos por si nos los encontramos formando parte de la documentación de algún programa.

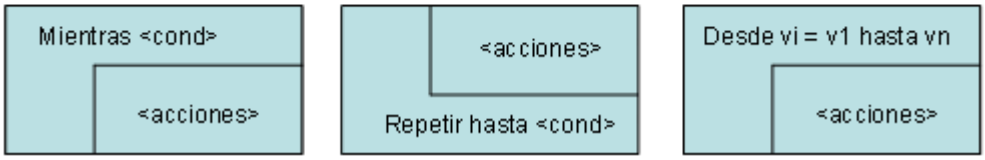
Símbolos empleados en los diagramas N-S

Los diagramas N-S pretenden resumir un algoritmo con una forma muy simple de representar cada uno de los pasos que se van a ejecutar de forma secuencial en sentido descendente y utiliza los siguientes estructuras para las sentencias condicionales y las repetitivas.

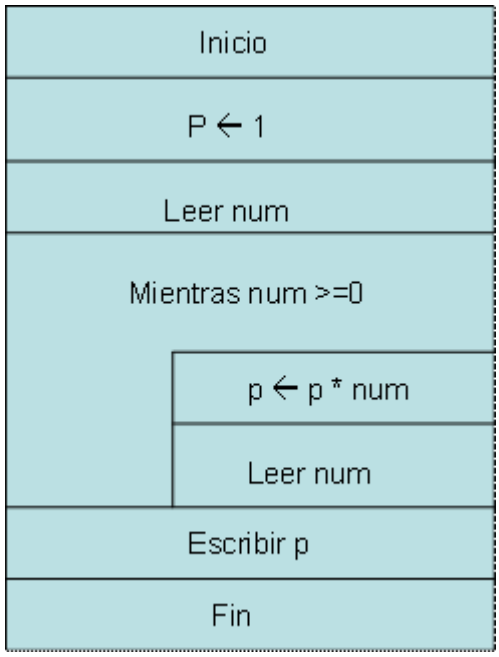
Condiciones:



Estructuras repetitivas:



Con este tipo de diagramas el algoritmo del apartado anterior quedaría representado de la siguiente forma:



Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

6.3. Pseudocódigo.

Unidad Didáctica IV

Pseudocódigo.

Por fin llegamos a la tercera forma de representar un algoritmo, que por otra parte es la más comúnmente usada. Es una forma intermedia entre el **lenguaje natural** y el **lenguaje de programación** para la descripción de la solución de un problema, que evita las ambigüedades del lenguaje natural y no entra en los detalles de **sintaxis** de un lenguaje de programación concreto.

- **Es un lenguaje de especificación de algoritmos**, pero **muy parecido a cualquier lenguaje de programación**.
- **Su traducción a cualquier lenguaje concreto es muy sencilla.**
- **No se rige por las normas de un lenguaje en particular,no hay que tener en cuenta la sintaxis concreta de cada sentencia, sino que podemos hacerlo casi como queramos, siempre que lo que escribamos no sea ambiguo y sea claro**.
- **Se centra más en la lógica del problema** que en los detalles de sintaxis.
- **No es tan visual como los diagramas de flujo o los diagramas N-S**.
- **Son bastante más fáciles de hacer y de corregir que las representaciones gráficas, ya que son sólo texto.**

El pseudocódigo también va a utilizar una serie de palabras clave o palabras especiales que va indicando lo que debe hacer el algoritmo.

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

6.3.1. Estructura de un algoritmo en pseudocódigo.

Unidad Didáctica IV

Estructura de un algoritmo en pseudocódigo

Hemos comentado que el pseudocódigo es parecido a cualquier lenguaje de programación, y por ello tiene una estructura más o menos definida, como los programas. ¿Qué partes componen esa estructura?

En pseudocódigo el algoritmo tiene tres partes: la **cabecera**, la **zona de declaración de constantes y variables** y el **cuerpo**.

- La **cabecera** contiene el nombre del algoritmo.
- La **zona de declaraciones** de variables y constantes establece qué variables y constantes vamos a usar, indicando junto al nombre el tipo de las mismas y el valor en el caso de las constantes.
- El **cuerpo** del algoritmo es el que contiene las instrucciones que indican las acciones a realizar por el programa. Comienza con la palabra Inicio y termina con la palabra Fin (o palabras similares).
- El aspecto final del algoritmo es algo similar al siguiente esquema, en el que los corchetes angulares indican que lo que va dentro se sustituirá por un nombre concreto, un tipo concreto, etc.

```
Algoritmo <nombre alg>
Const  <nombre>:<tipo>, <nombre>: <tipo>,...
Var    <nombre>:<tipo>, <nombre>: <tipo>,...
Inicio
  <Instrucciones>
Fin
```

En el cuerpo, para que quede más legible, hay que usar la indentación y si es necesario hay que usar comentarios identificados como tales por algún símbolo, tal como llaves, o entre /* */, que es ya casi un estándar.

Las distintas estructuras de control de flujo que se pueden usar en un algoritmo son las siguientes:

Estructura de Control	Descripción
Inicio y Fin	Por donde empieza y acaba el algoritmo.
Si <condición> Entonces <acciones1> Si no <acciones2> Fin-Si	Ejecución condicional de la acción.
Mientras <condición> hacer <acciones> Fin-Mientras	Repetición de la acción mientras que la condición sea verdadera.
Repetir <acciones> hasta <condición>	Repetición de la acción hasta que la condición sea verdadera.
Desde <Variable = valor inicial> hasta <valor final> paso <incremento> hacer < acciones> Fin-Desde	Repetición de la acción dependiendo de una variable de control que empieza tomando el valor inicial, y tras cada vuelta se incrementa sumándole incremento, hasta que se alcance o supere el valor final.
Según Sea <expresión> Caso <valor1>: <acciones1> Caso <valor2>: <acciones2>	Se comprueba el valor de la expresión, y según sea su valor, se ejecuta la acción que le corresponda.

… Caso <valorN>: <accionesN> [Otro Caso : <accionesPorDefecto>] Fin-Según	Los corchetes normales indican que el tro caso es opcional, puede no ponerse.

AUTOEVALUACIÓN:

Podemos afirmar que el pseudocódigo es:

- a) Una forma intermedia entre el lenguaje natural y el lenguaje de programación para la descripción de la solución de un problema, que evita las ambigüedades del lenguaje natural y no entra en los detalles de sintaxis de un lenguaje de programación concreto.
- b) Una forma intermedia entre el lenguaje natural y el lenguaje de programación para la descripción de la solución de un problema, que evita las ambigüedades del lenguaje natural y que sigue la sintaxis de un lenguaje de programación concreto.
- c) Un tipo de lenguaje de programación con una sintaxis muy simple que permite la representación de una manera cómoda de un programa.
- d) Todas las anteriores son correctas.

Comprobar

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

7. Estructuras básicas de control del flujo.

Unidad Didáctica IV

Estructuras básicas de control del flujo

Víctor ha entendido que cada uno de sus compañeros utiliza la representación de algoritmos que prefiere y que eso no debe ser ningún impedimento para que cualquiera pueda leer su algoritmo, entenderlo y codificarlo a cualquier lenguaje de programación. La verdad es que esto le está gustando a Víctor, se siente parte de un equipo y cree que puede llegar a trabajar como cualquiera de sus compañeros. Jesús le ha dicho en alguna ocasión que un buen programador es el que puede hacer algoritmos eficientes y eso se consigue dominando el control de ejecución del algoritmo, para lo que es necesario conocer perfectamente las estructuras de control de flujo.

Hemos mencionado en el apartado anterior las estructuras de control de flujo que se usan en pseudocódigo. ¿Son las únicas posibles? ¿Son todas ellas imprescindibles?

En realidad no todas son imprescindibles, pero sí representan a casi todas las posibles, al menos a todas las que son recomendables. Cualquier problema se resuelve haciendo uso de un conjunto limitado de estructuras de **control del flujo** de ejecución de las sentencias. Aquí presentamos estas estructuras. Matemáticamente es posible demostrar que **para todo problema con solución** puede describirse un algoritmo que lo solucione **usando sólo las tres estructuras de control de flujo básicas**. Esas estructuras son las siguientes:

Secuencial. Consiste en escribir o representar las sentencias una detrás de otra justo en el orden en que deben ejecutarse.

- **Condicional o selectiva.** Consiste en la ejecución o no de unas sentencias dependiendo del **valor de verdad** de una condición. En todos los lenguajes existen distintas sentencias correspondientes a la estructura condicional.
- **Cíclica, iterativa o repetitiva.** Consiste en ejecutar repetidamente unas sentencias mientras se cumpla una condición. También existen en todos los lenguajes distintas sentencias correspondientes a la estructura cíclica.

Jesús dice que cada programador luego adopta las estructuras y técnicas que mejor se acoplan a su forma de entender la programación, aunque sin renunciar al resto. Añade que el problema también influye en ocasiones para decidir cuál es la estructura a utilizar, incluso una estructura condicional mejor que otra. Lo mismo ocurre con las estructuras cíclicas.

El hecho de que los lenguajes incluyan varias sentencias de cada tipo responde más al deseo de proporcionar una variedad de sentencias que se adapten mejor a los distintos casos posibles y que faciliten la programación, que al hecho de que sean realmente necesarias. De hecho, sería posible construir cualquier programa usando un solo tipo de sentencia condicional y un solo tipo de sentencia cíclica.

PARA SABER MÁS.
Sitio muy interesante sobre programación estructurada. En esta página concretamente hace un breve comentario sobre los tipos de estructuras de control de flujo que podemos encontrar.
Definición de las Estructuras Básicas de Control. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web).

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

7.1. Estructura secuencial

Unidad Didáctica IV

Estructura secuencial

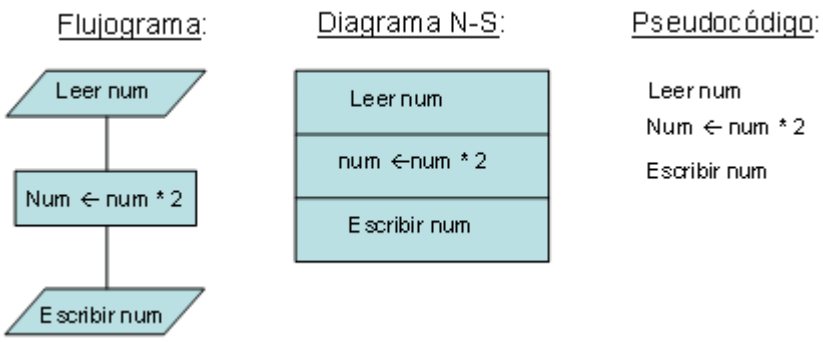
Siempre que tengas que indicar una serie de pasos que deben darse en el orden que se indican, deberás usar esta estructura.

Es cuando una instrucción sigue a otra en secuencia, escritas en el orden en que van a ejecutarse.

En pseudocódigo es la estructura:

```
Sentencia1
Sentencia2
Sentencia3
...
sentenciaN
```

Como ejemplo puedes ver cómo se representa la misma estructura secuencial en los tres sistemas de representación de algoritmos que hemos visto:



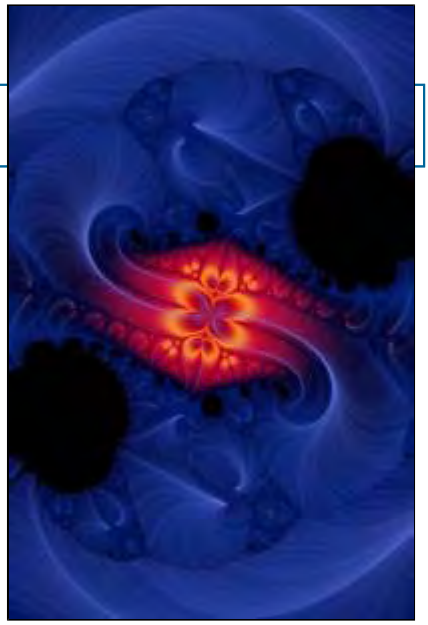
Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

7.2. Estructura condicional, selectiva o alternativa.

Unidad Didáctica IV

Estructura condicional, selectiva o alternativa

¿Tenemos que comprobar alguna condición antes de decidir qué sentencia es la que debemos ejecutar? En ese caso la estructura condicional es la adecuada.



Se evalúa la condición y en función del resultado se ejecuta un conjunto de instrucciones u otro.

Hay tres tipos de sentencias selectivas:

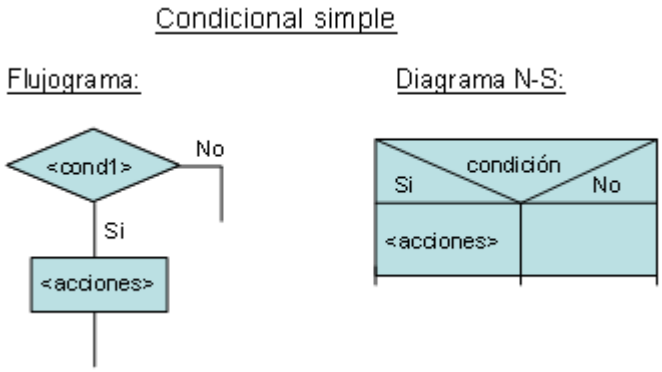
- **Condicional simple.**
- **Condicional doble.**
- **Selectiva múltiple**. En el caso de la selectiva múltiple, lo que se evaluará es una expresión, que dará un valor de un tipo discreto (dado un valor, sabemos cual es el siguiente). Dependiendo del valor resultante, se ejecutará una u otra sentencia. Es decir, cada uno de los valores posibles tendrá asociada su propia sentencia, que será la única que se ejecutará cuando la expresión tome ese valor.

7.2.1. Condicional Simple.

Unidad Didáctica IV

Condicional Simple

Si necesitas hacer alguna sentencia sólo en el caso de que se cumpla alguna condición, el condicional simple es la estructura adecuada.



Evaluamos la condición y si es verdadera ejecutamos el conjunto de sentencias asociadas al entonces, y si es falso, no hacemos nada y continuamos con la sentencia que haya a continuación.

En pseudocódigo es la estructura:

```
Si <condición>  entonces
        <acciones>
Fin-Sí;
```

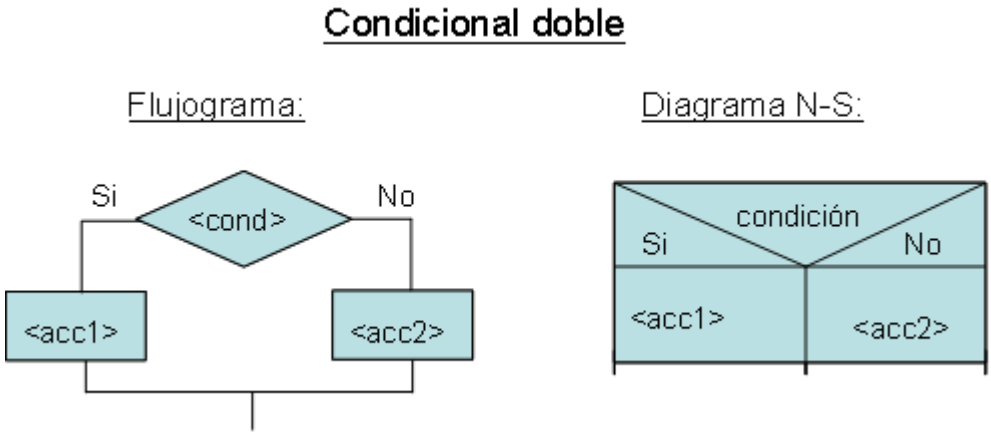
7.2.2. Condicional Doble.

Unidad Didáctica IV

Condicional Doble

Si tienes que elegir entre dos posibles alternativas, dependiendo de que se cumpla o no alguna condición, debes utilizar el condicional doble.

Se evalúa la condición y si es verdad se ejecutan el conjunto de acciones asociadas a la parte entonces, y si es falso se ejecutan el conjunto de acciones asociadas a la parte sino (se elige uno de los dos caminos y se descarta el otro).



La condición se evalúa una única vez, por lo que las acciones que correspondan se ejecutarán una única vez.

En pseudocódigo esta estructura es:

```
Sí <condición> entonces
        <acciones1>
Si no
        <acciones2>
Fin-Sí;
```

7.2.3. Alternativa Múltiple o Selectiva Múltiple

Unidad Didáctica IV

Alternativa Múltiple o Selectiva Múltiple

Si tienes un caso en el que hay múltiples valores posibles para una expresión, y cada uno de ellos requiere efectuar un conjunto propio de acciones, la estructura más indicada será la alternativa o selectiva múltiple.

Se evalúa una expresión que puede tomar múltiples valores. Según el valor que la expresión tenga en cada momento (según el caso de que se trate) se ejecutan las acciones correspondientes al valor.

En realidad equivale a un conjunto de condiciones anidadas. En cualquier lenguaje suele usarse con las palabras Case o Switch.

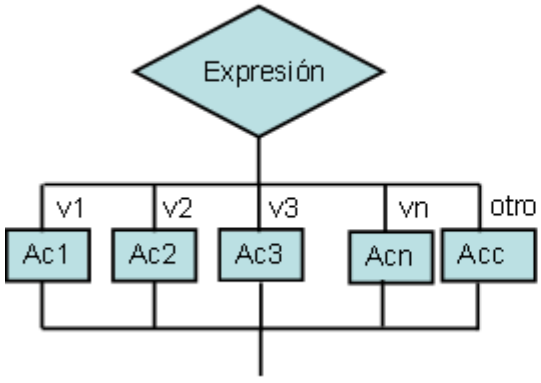
Es la estructura:

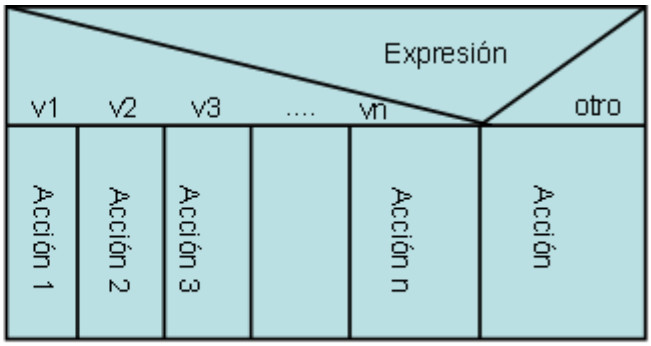
```
Según Sea <expresión>
     Caso <Valor1>: <acciones1>
     Caso <valor2>: <acciones2>
     ...
     Caso <valorN>: <accionesN>
     [<Otro Caso>: <accionesPorDefecto>]
Fin-Según
```

Otro Caso: Las acciones asociadas a la etiqueta <Otro Caso> se ejecutan cuando la expresión no toma ninguno de los valores previstos en las demás etiquetas, y suele ser opcional. **otherwise, else, default** son palabras que se usan frecuentemente para esta alternativa.

Los valores que se pueden poner como etiquetas (como alternativas a comprobar) van a depender de los lenguajes. Algunos lenguajes permiten usar cualquier expresión que devuelva un tipo ordenado y discreto, y en las etiquetas permiten poner valores individuales o rangos de valores, e incluso condiciones que determinen rangos de valores.

Otros, como es el caso de Java, sólo permite usar expresiones de tipo entero, y como etiquetas sólo valores individuales, aunque queramos que se hagan las mismas acciones para varios de ellos.





Como casi siempre, un ejemplo puede ayudar bastante a entender esto.

Ejemplo:

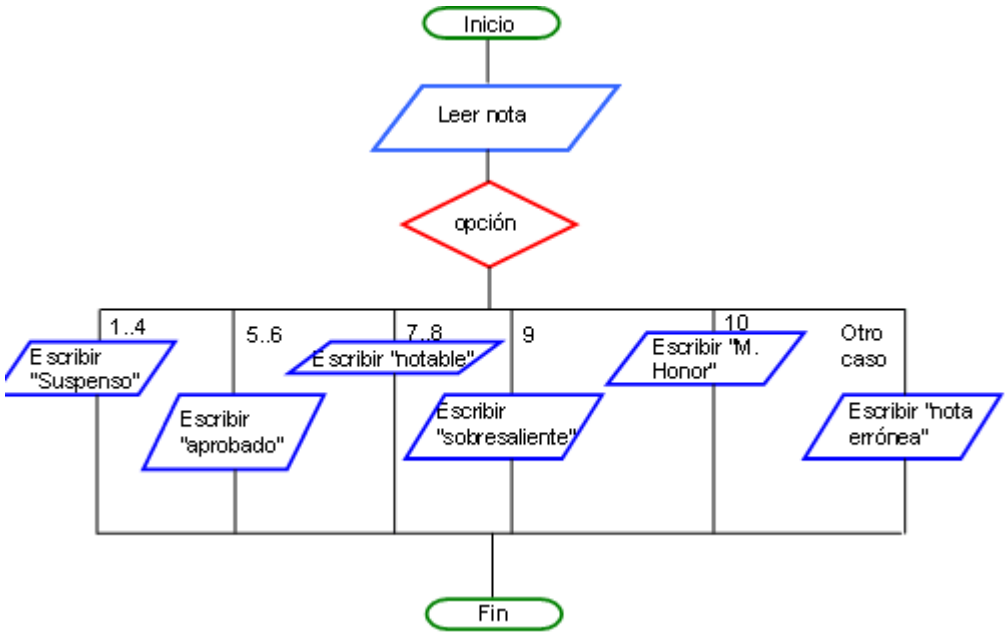
Hacer un programa que pueda dibujar una recta, un punto o un rectángulo.

```
Algoritmo Dibujo
Var op: carácter
Escribir ("Introduce una opción"
         1.-Punto
         2.-Recta
         3.-Rectángulo")
Leer op
Según Sea op
   "1":       /*dibujar punto*/
         .........
   "2":       /*dibujar recta*/
         ..........
   "3":       /*dibujar rectángulo*/
         ..........
   "otro": escribir "opción errónea"
Fin-Según
```

Otro Ejemplo:

Hacer un algoritmo para leer una nota numérica y escribir en pantalla la calificación asociada:

```
 Algoritmo LeerNotaEscribirCalificación
Var nota: entero
Leer nota
Según Sea nota
   1..4: escribir ("suspenso")
   5..6: escribir ("aprobado")
   7..8: escribir ("Notable")
   9: escribir ("Sobresaliente")
   10: escribir ("Matricula de honor")
   Otro Caso: escribir ("Nota errónea. Sólo se admiten notas entre 1 y 10")
Fin-Según
```



En algunos lenguajes se permite poner una condición que determine un rango. Algo como lo que se indica a continuación:

```
Según sea nota
        nota >=1 y nota <=4: escribir "suspenso"
...
```

DEMO: Puedes ver la elaboración de este algoritmo en la siguiente simulación en la que cada paso es comentado para que comprendas lo que piensa el programador al diseñar un algoritmo.

En pseudocódigo supondremos que no se pueden poner condiciones de este tipo en las sentencias selectivas múltiples, ya que son infrecuentes en los lenguajes, y además Java que es el lenguaje que usaremos en el curso, no las permite. Así nos vamos acostumbrando.

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

8. Estructuras repetitivas, cíclicas o iterativas.

Unidad Didáctica IV

Estructuras repetitivas, cíclicas o iterativas.

Jesús dice que todos los programadores entienden que el diseño de un algoritmo requiere de un profundo conocimiento del problema unido a un moderado dominio de las estructuras de control de flujo, pero hay situaciones en las que se precisa de cierta creatividad mediante la que deciden en qué momento es más conveniente utilizar una estructura frente a otra.

Sin duda las estructuras repetitivas o cíclicas son las que más se ajustan a esta afirmación, ya que repetir algunos de los pasos reduce el tamaño del algoritmo y lo hace más eficiente. Estas estructuras las utilizamos casi a diario, con el fin de hacer más sencilla una tarea, pero en ocasiones es el único modo de hacer determinadas cosas.

Son muchas las situaciones en las que para solucionar un problema es necesario repetir una serie de pasos hasta que se consiga un determinado resultado, o mientras se cumpla una determinada condición.
Piensa por ejemplo en el profesor que corrige y evalúa los exámenes de los alumnos de una clase. Habrá una serie de tareas que tendrá que repetir para cada alumno (coger el examen, corregir cada pregunta, puntuarlas, sumar la puntuación del examen, anotar la nota en su cuaderno, guardar el examen)Y todos esos pasos deberá repetirlos mientras que queden exámenes de alumnos (o hasta que se agoten los exámenes, que es otra forma de decir lo mismo)

Las estructuras repetitivas son aquellas que contienen un bucle (conjunto de instrucciones que se repiten un número finito de veces). Cada repetición del bucle se llama iteración.

- Todo bucle tiene que llevar asociada una condición, que es la que va a determinar cuando se repite el bucle.
- Después de cada iteración se vuelve a evaluar la condición, y si sigue cumpliéndose, se vuelven a ejecutar todas las sentencias del bucle.
- En las instrucciones del bucle debe haber alguna o algunas que modifiquen el valor de la condición para evitar que entremos en un bucle infinito.
- Si la condición es verdadera cuando entramos, y no hay nada que la modifique, siempre seguirá siendo verdadera y volveremos a realizar otra iteración, hasta el infinito.

Los tipos de bucles son:

Mientras - hacer --> While - do
- **Repetir - hasta --> Repeat - until**
- **Desde --> For**

AUTOEVALUACIÓN

Las sentencias de control del flujo que permiten repetir una serie de acciones dependiendo del cumplimiento de una condición son del tipo:

a) Incondicional

b) Cíclica o repetitiva
c) Condicional o selectiva
d) Secuencial

Comprobar

Los bucles tipo "repeat" son más adecuados:

a) Cuando sabemos que las sentencias del bucle siempre se van a tener que ejecutar al menos una vez
b) Cuando queremos comprobar la condición antes de ejecutar el bucle, y si no se cumple, ni siquiera se ejecutarían las sentencias del bucle una vez.
c) Cuando conocemos cuantas veces hay que ejecutar las sentencias que contiene el bucle o los valores inicial y final para los que se ejecuta.
d) Cuando queremos escribir un ciclo while de forma abreviada

Comprobar

8.1. Estructura repetitiva tipo mientras-hacer (while-do)

Unidad Didáctica IV

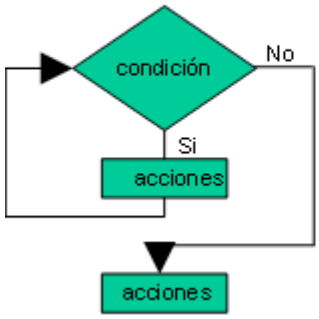
Estructura repetitiva tipo mientras-hacer (while-do)

¿Cuando será apropiado usar este tipo de bucle en vez de otro? Si es necesario comprobar la condición antes de hacer ninguna de las acciones que deben repetirse, ésta es la estructura adecuada.

La condición del bucle se evalúa al principio, antes de entrar en él. Si la condición es verdadera, comenzamos a ejecutar las acciones del bucle y después de la última volvemos a evaluar la condición para ver si hay que volver a ejecutar el bucle de nuevo. En el momento en el que la condición sea falsa nos salimos del bucle y ejecutamos la siguiente sentencia posterior al bucle.

Al evaluarse la condición al principio, antes de entrar en el bucle, **si la condición ya es falsa la primera vez, no entraremos en el bucle, y no se ejecutará ninguna de las acciones que contiene ninguna vez**. Por tanto usaremos obligatoriamente este tipo de bucle en el caso de que exista la posibilidad de que el bucle no deba ejecutarse ninguna vez.

FLUJOGRAMA:



DIAGRAMAS NS:

La estructura en pseudocódigo es:

```
Mientras <condición> hacer
         <acciones>
Fin-Mientras
```

Una vez más, un ejemplo puede ayudar.

Ejemplo:

Mira la descripción del siguiente algoritmo:

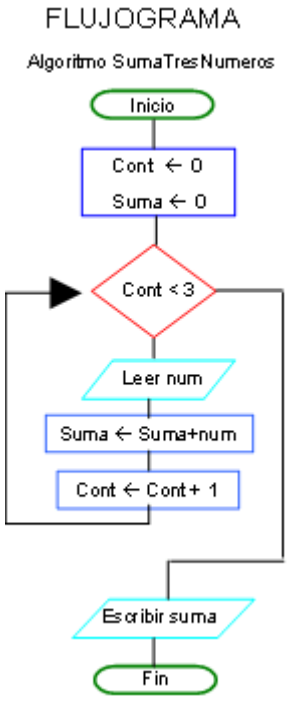
Queremos hallar el producto de varios números positivos introducidos por teclado. El proceso debe terminar cuando se introduzca un número negativo.

En un apartado anterior se describió la solución en diagrama de flujo,y ahora lo hacemos con pseudocódigo y usando la estructura cíclica mientras-hacer:

```
Algoritmo Producto
```

```
Var
  P, num: entero
Inicio
  P ← 1
  Leer num
  Mientras num >=0 hacer
    P ← P * num
    Leer num
  Fin-Mientras
  Escribir (P)
Fin
```

Otro Ejemplo:



Leer 3 números y dar su suma:

PSEUDOCÓDIGO

```
Algoritmo SumarTresNumeros
Var
        Cont, Suma, num: entero
Inicio
        Cont ← 0
        Suma ← 0
        Mientras Cont ← 3
                 Leer num
                 Suma ← Suma + num
                 Cont ← Cont + 1
        Fin mientras
        Escribir (''la suma es'', Suma)
Fin
```

DEMO: En esta simulación puedes observar cómo se ejecutaría el algoritmo con un

ejemplo.

AUTOEVALUACIÓN:

Dado el siguiente algoritmo en pseudocódigo señala el resultado correcto

```
x ← 1
R ← 0
Mientras x <=4
  R ← x * R
  x ← x + 1
Fin-Mientras
```

- a) R=256
- b) R=0
- c) R=64
- d) R=32

Comprobar

Dado el siguiente algoritmo en pseudocódigo señala el resultado correcto:

```
x ← 1
R ← 1
Mientras x <=3
  R ← x * R
  x ← x + 1
Fin-Mientras
```

- a) R=2
- b) R=0
- c) R=6
- d) R=18

Comprobar

Dado el siguiente algoritmo en pseudocódigo señala el resultado correcto:

```
Cont ← 0
Suma ← 0
Mientras cont < 3
  Suma ← Suma + Cont
  Cont ← Cont + 1
Fin-Mientras
```

- a) Suma=0
- b) Suma=1

c) Suma=2

d) Suma=3

Comprobar

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

8.2. Estructura repetitiva tipo repetir-hasta (repeat-until)

Unidad Didáctica IV

Estructura repetitiva tipo repetir-hasta (repeat-until)

¿Piensas que esta estructura será adecuada para los mismos casos que la estructura cíclica tipo hacer-mientras?

Según se mire. Es posible construir una sentencia hacer-mientras usando otra tipo repetir-hasta, pero ésta última es muy adecuada en aquellos casos que queremos garantizar que las sentencias del bucle se van a ejecutar al menos una vez.

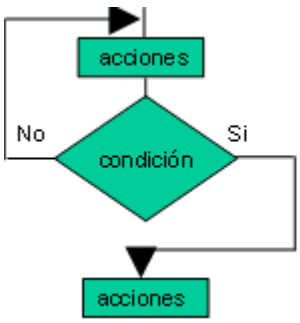
Si por ejemplo el bucle tiene que repetir la lectura de un número hasta que tome un valor mayor que 5, sabemos que siempre vamos a tener que hacer al menos la primera lectura. Si es la adecuada, no se repetirá más la lectura, pero por lo menos un número seguro que necesitamos leer.

En este tipo de bucles, se repiten las sentencias que incluye hasta que la condición sea verdadera, que es lo mimo que decir que se repite mientras la condición sea falsa. La condición se evalúa siempre al final del bucle, si es falsa volvemos a ejecutar las acciones, si es verdad se sale del bucle.

Como la condición se evalúa al final, incluso aunque la primera vez ya sea verdadera, habremos pasado al menos una vez por el bucle y realizado una iteración.

No obstante, algunos lenguajes, como Java, no tienen exactamente esta estructura, si no una parecida, que podríamos llamar "repetir-mientras". (Do-While) Por lo demás es igual. Las acciones del bucle se hacen al menos una vez, y al final del bucle se evalúa la condición, y si es cierta, se vuelve a repetir el bucle. Si es falsa, se sale del bucle. Aunque la condición fuera falsa de entrada, la primera vez de todas formas se ejecutaría. Es decir, funciona exactamente igual que un repeat-until si ponemos justamente la condición contraria. (sería lo mismo decir "repite las acciones hasta que n >10" que decir "haz las acciones mientras n <=10")

Naturalmente se podría hacer lo mismo que hace cualquier bucle repetir - hasta usando bucles mientras - hacer.



La estructura repetir-hasta, en pseudocódigo sería:

```
Repetir
  <acciones>
hasta <condición>
```

Como siempre, es oportuno verlo con un ejemplo.

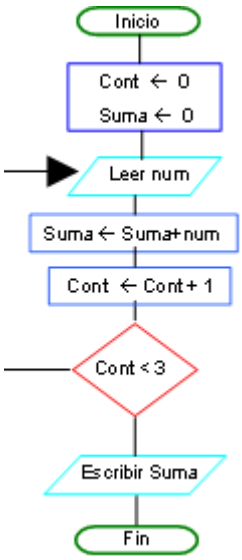
Ejemplo:

Otra forma de conseguir el mismo resultado para el algoritmo del punto anterior, que leía números positivos y los multiplicaba hasta que se introducía un negativo sería usar la estructura repetir-hasta:

```
 Algoritmo Producto
Var
  P, num: entero
Inicio
  P ← 1
  Repetir
    Leer num
    Si num >=0
      Entonces P ← P * num
    Fin-Si
  Hasta num < 0
  Escribir p
Fin
```

Otro Ejemplo:

Leer 3 números y dar su suma:




```
 Algoritmo SumarTresNumeros
Var Cont, Suma, num: entero

Inicio
        Cont ← 0
        Suma ← 0
        Repetir
          Leer num
          Suma ← Suma + num
          Cont ← Cont + 1
        Hasta Cont >= 3
        Escribir "la suma es", Suma
Fin
```

Aunque parece hacer lo mismo que el ejemplo usando hacer-mientras del apartado anterior, no son totalmente iguales, ya que si por ejemplo Cont se inicializa a 3 en ambos, el primero (mientras) no se ejecutaría ninguna vez, mientras que el segundo (repetir) se ejecutaría la primera vez siempre.

Para conseguir el mismo efecto en el primer caso, bastaría con copiar todas las sentencias del bucle justo delante del mismo, para que se ejecutaran de forma incondicional la primera vez antes de comprobar la condición. En ese caso el efecto sería el mismo, pero el esfuerzo de escribir el bucle por duplicado no merece la pena.

AUTOEVALUACIÓN

Dado el siguiente algoritmo en pseudocódigo señala el resultado correcto:

```
suma ← 1
cont ← 1
Repetir
  num ← 2 * cont
  suma ← suma + num cont
  cont ← cont + 1
Hasta cont >= 3
```

- a) Suma=2
- b) Suma=3
- c) Suma=6
- d) Suma=7

Comprobar

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

8.3. Estructura repetitiva tipo for.

Unidad Didáctica IV

Estructura repetitiva tipo for

En los casos anteriores tenemos que repetir un conjunto de sentencias una serie de veces, pero no tenemos en cuenta si conocemos de antemano o no el número de veces que se tiene que ejecutar el bucle. ¿Es posible simplificar las cosas si sabemos que un bucle se debe repetir, por ejemplo, exactamente 7 veces? Justamente es lo que hace el **ciclo tipo for.**

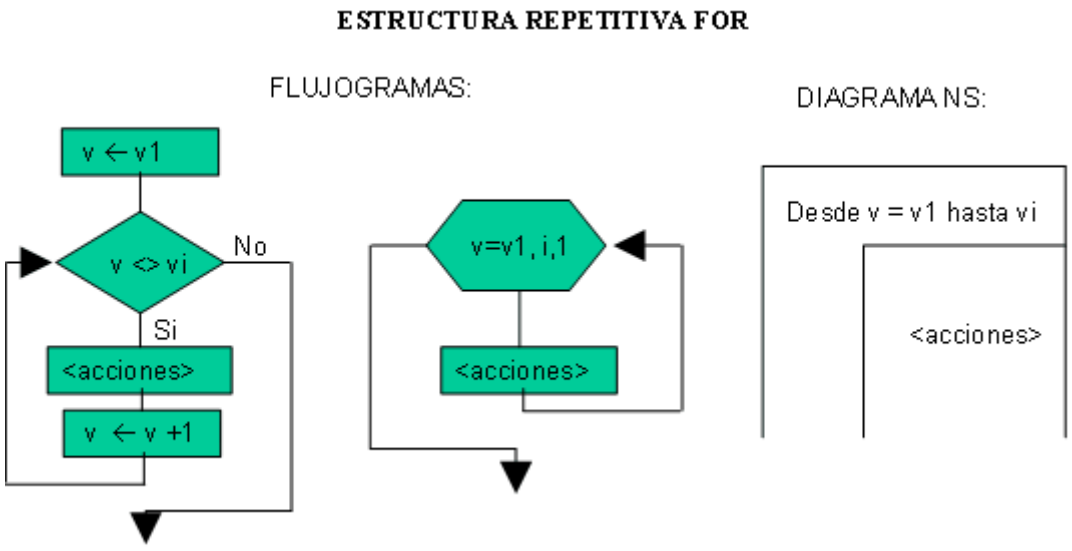
Este tipo de bucles se utiliza cuando se sabe ya antes de ejecutar el bucle el número exacto de veces que hay que ejecutarlo. Para ello el bucle llevará asociada una variable que denominamos variable índice, a la que le asignamos un valor inicial y determinamos cual va a ser su valor final. La variable índice se va a incrementar o decrementar en cada iteración del bucle en un valor constante, pero esto se va a hacer de manera automática al terminar cada iteración, (el programador no se tiene que ocupar de incrementar o decrementar esta variable en cada iteración), sino que va a ser una operación implícita (lo hace internamente la propia estructura).

Por tanto en cada iteración del bucle, la variable índice se actualiza automáticamente y cuando alcanza el valor que hemos puesto como final se termina la ejecución del bucle.

En algunos lenguajes como C o Java, esta estructura se ha modificado para convertirla en una versión reducida de un bucle tipo mientras-hacer, introduciendo algunas diferencias, como veremos más adelante, al introducirnos en el uso de Java.

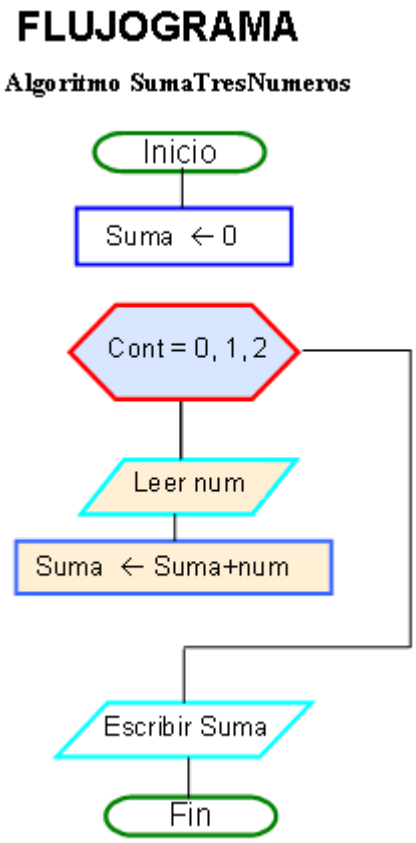
La estructura en pseudocódigo es:

```
Desde <Variable=valor inicial> hasta <valor final> paso <incremento>
      hacer <acciones>
Fin-Desde
```



El mismo ejemplo de la suma de tres números, podemos verlo con una estructura tipo for.

Ejemplo:



Leer 3 números y dar su suma:

PSEUDOCÓDIGO

```
Algoritmo SumarTresNumeros

Var Cont, Suma, num: enteros
Inicio
        Suma ← 0
        Desde Cont = 0 hasta 2 paso 1
                Leer num
                Suma ← Suma + num
        Fin-Desde
        Escribir "la suma es", Suma
Fin
```

9. Ejemplos de algoritmos resueltos.

Unidad Didáctica IV

Ejemplos de algoritmos resueltos

A Víctor le están enseñando entre todos a confeccionar algoritmos eficientes. Pero quizás su compañera más cercana es Carmen. Han decidido entre todos ponerle una tarea, que consiste en una serie de ejercicios para que practique durante el fin de semana. Víctor no está muy contento con esto de que le hagan trabajar durante el fin de semana y como un extra respecto al resto de compañeros, pero sabe que la única forma de hacer bien los algoritmos es con la práctica, y como dice María; "Quien mejor hace los algoritmos es el que más ha hecho, ya que es la mejor forma de llegar a dominar la técnica". Carmen entiende a Víctor y se da cuenta de lo que se le viene encima, así que ha decidido ayudarle en su tarea, le ha dejado unos ejercicios resueltos que ella hizo durante el pasado curso en el módulo de Programación en Lenguajes Estructurados, cuando empezaba con la programación. Le indica que cada uno de estos ejercicios trata alguno de los aspectos vistos en el tema, por lo que es muy recomendable que los haga por sí mismo y que probablemente cada uno le facilite la confección del siguiente.

Se pretende que tengas a tu disposición algunos **ejemplos de diagramas de flujo y pseudocódigos que representan algunos algoritmos**, para que te familiarices con su uso, y te sirvan de guía a la hora de resolver tus propios problemas y representar la solución de forma clara. Puesto que la finalidad es comprender y familiarizarse con el uso de las estructuras básicas de control de flujo, la funcionalidad práctica de los ejemplos es limitada, y fundamentalmente se exponen ejemplos que requieren efectuar cálculos matemáticos, ya que en ellos el problema a resolver suele estar más claro.

Hay que dejar claro que incluso en estos ejercicios, bastante básicos, es posible encontrar varias soluciones, que aunque parecidas tengan ligeras diferencias entre sí. Si varios de vosotros intentáis resolver estos problemas, u otros similares, casi con toda seguridad que las soluciones encontradas tendrán diferencias, que en algún caso pueden ser incluso significativas. En la mayoría de los programas que se abordan (en el curso y en la vida real) normalmente no existe una única solución válida. Nuestra obligación es encontrar la más sencilla y eficiente. Pero eso se consigue sólo con la práctica, por lo que una tarea importante en esta unidad será que vosotros os entrenéis en la descripción de algoritmos que solucionen problemas.

Estos ejemplos por ser los primeros, llevan muchos comentarios (que no tienen efecto en su funcionamiento, y que hemos diferenciado por ello en otro color), pero dada la simplicidad de estos algoritmos, en condiciones normales no deberían llevar ni tantos comentarios ni tan extensos.

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

9.1. Cálculo del perímetro y superficie de un círculo, leído el valor del radio.

Unidad Didáctica IV

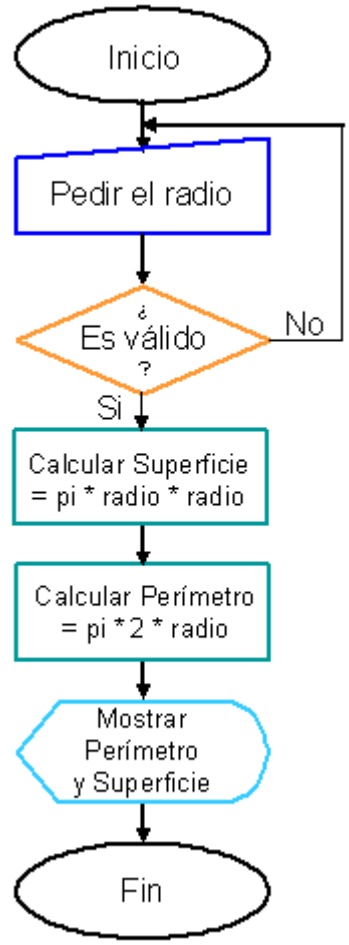
Cálculo del perímetro y superficie de un círculo, leído el valor del radio

***Carmen** le recomienda que en este ejercicio preste especial atención a la entrada del valor del radio y en los cálculos de las fórmulas.*

Queremos calcular la superficie y el perímetro de un círculo, una vez introducido por teclado el valor del radio, y escribir los resultados. Se supone conocido &pi = 3.1416, y que las fórmulas para la superficie y el perímetro son:

Sup = &pi r^2 y Per = 2 &pi r siendo r el valor del radio.

FLUJOGRAMA



PSEUDOCÓDIGO

```
Algoritmo Circulo

      /* Declaración de la constante PI. Se declara como constante porque
su valor no va a cambiar.*/
Const
         PI = 3.1416
      /* Declaración de variables. De tipo real para que admitan decimales.*/
```

```
Var
        Radio, Superficie, Perímetro : real
Inicio
              /* Pide que se facilite el valor del Radio */
        Radio ← Valor leído por teclado
              /* Comprueba si el valor del Radio es correcto */
        Si el valor del Radio no es un entero positivo volver a pedirlo.
En caso contrario continuar con el paso siguiente.
              /* Calculamos la superficie y el perímetro según la fórmula */
        Superficie ← PI * Radio * Radio
        Perímetro ← 2 * PI * Radio
              /* Escribimos los resultados */
        Escribir ("El círculo de radio", Radio, "tiene una superficie de", Superf
        "y un perímetro de " , Perímetro)
Fin
```

DEMOS: Puedes ver dos ejemplos de la ejecución de este algoritmo en las siguientes simulaciones:

- **En el primero de los ejemplos el funcionamiento es inmediato ya que se introduce valor 4 para el radio del círculo.**
- **En el segundo el radio que se introduce es erróneo y debe volver a pedirlo.**

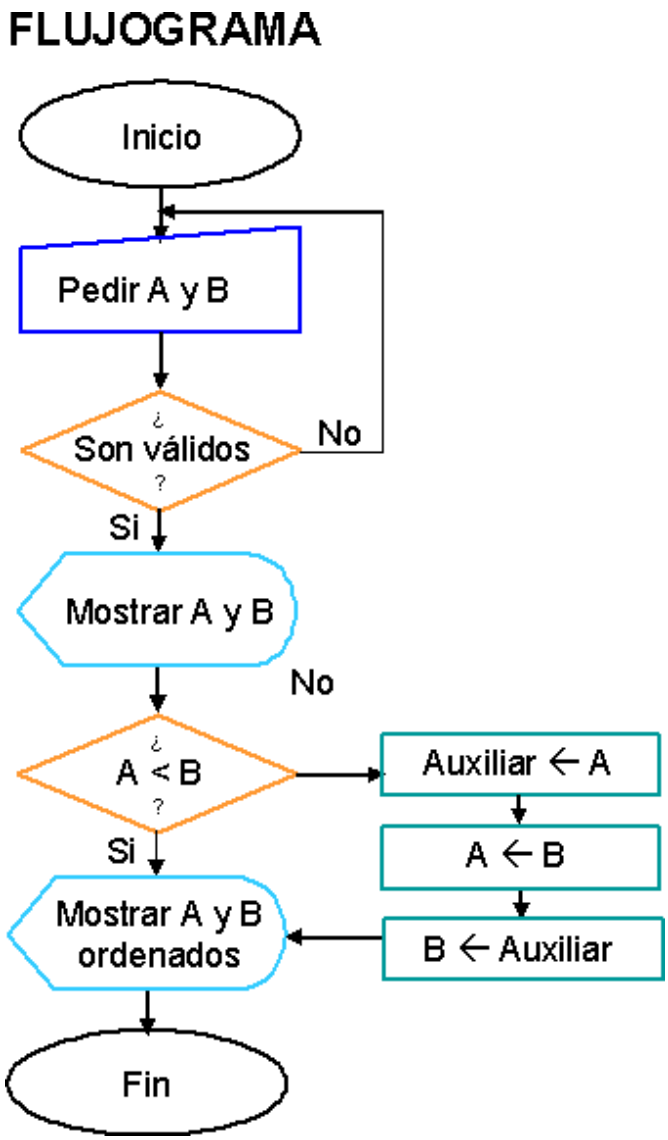
9.2. Intercambio del valor de dos variables.

Unidad Didáctica IV

Intercambio del valor de dos variables

Este ejercicio es uno de los más utilizados, ya que el intercambio de los valores de dos variables es algo muy habitual en cualquier programa y es importante abordarlo de forma correcta con una auxiliar.

Se trata de una operación muy básica, pero frecuente. Suponemos que tenemos dos variables de tipo entero, A y B, a las que se les ha dado un valor cualquiera desde teclado, y que queremos intercambiar sus valores de forma que en A siempre quede el menor de los dos valores introducidos. Como salida, debe escribirse el valor que tenían inicialmente A y B y de nuevo los valores de A y B una vez ordenados.



PSEUDOCÓDIGO:

```
Algoritmo IntercambioVariables
```

```
Var A, B, Auxiliar : entero
      /* Auxiliar se usa para poder intercambiar los valores sin perder ninguno.*/
Inicio
         Leer A
         Leer B
     /* Al escribir, separamos entre comas los distintos elementos que
queremos escribir. Lo que va entre comillas es un literal que se
escribe tal y como está. Lo que no lleva comillas, es una variable,
 constante o expresión que se sustituye al escribir por su valor */
         Escribir ("Inicialmente A= ", A , " y B= ", B)
     /* Si A es mayor que B, se intercambian sus valores, y si no lo es,
no es necesario hacer nada especial, ya están ordenados */
         Si (A > B) Entonces
                  Auxiliar ← A        /* Se copia en Auxiliar el valor de A */
                  A ← B    /* Se escribe en A el valor de B, que sustituye a su a
                  B ← Auxiliar        /* Se escribe en B el valor de Auxiliar, que es
         Fin-Si
Escribir ("Después de ordenar  A= ", A , " y B= ", B)
Fin
```

9.3. Calcular la media de varias notas leídas desde teclado.

Unidad Didáctica IV

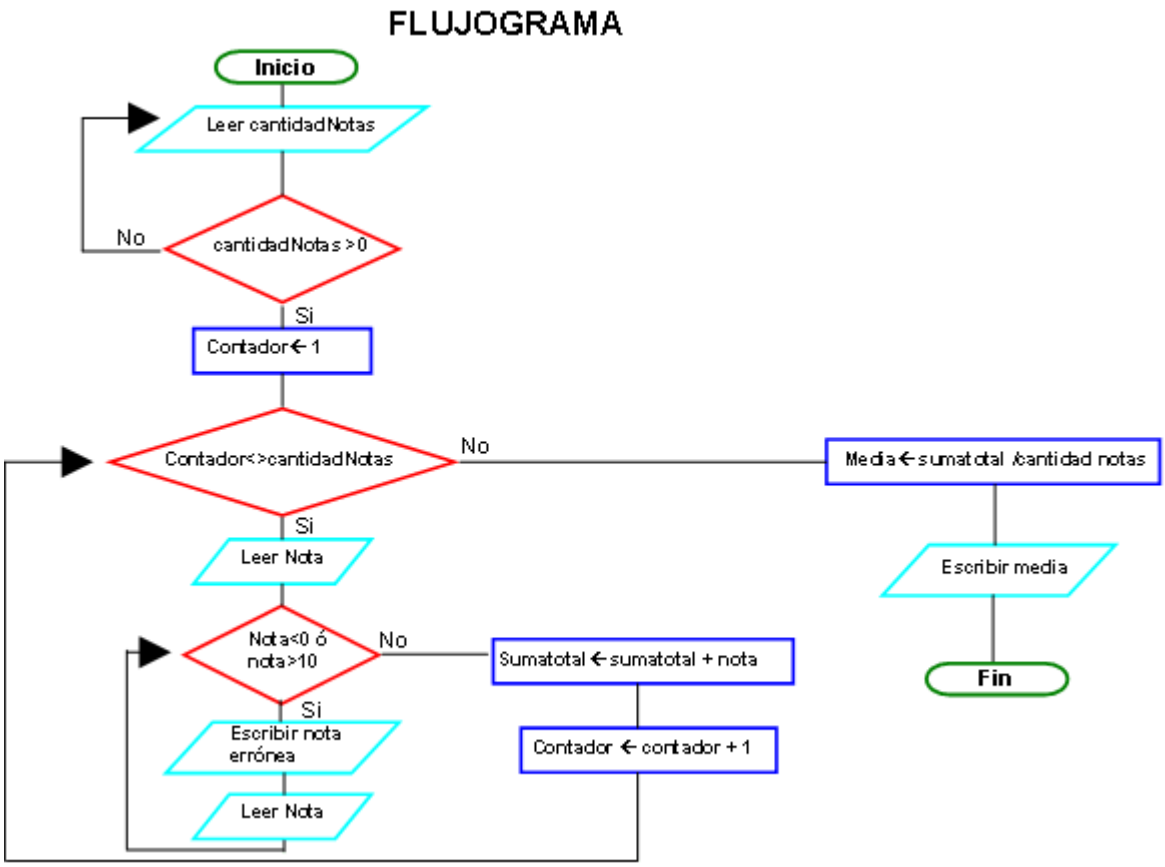
Calcular la media de varias notas leídas desde teclado

Aquí lo importante es hacer un correcto uso de una estructura repetitiva, ya que no sabemos el número de notas que se introducen. También es importante una correcta devolución de soluciones.

Queremos calcular la media de varias notas que se introducen desde teclado. A priori no sabemos cuántas notas van a leerse, por lo que lo primero que haremos en nuestro algoritmo es preguntarlo (leer un número positivo que nos indique cuántas son). A continuación las leeremos, descartando los valores que no sean correctos, calcularemos la media y escribiremos su valor en pantalla.

PSEUDOCÓDIGO:

```
Algoritmo MediaNotas

Var
         cantidadNotas, contador: entero
        nota, sumaTotal, media : real
      /* La media y las notas pueden  tener decimales */
Inicio
         /* Se repite la lectura de la cantidad de notas hasta que sea
         un valor positivo */
         Repetir
                  Leer cantidadNotas
         Hasta  cantidadNotas > 0
         sumaTotal ← 0          /* Se inicializa sumaTotal al valor cero */
               /* Se hace un bucle que se repite tantas veces como se indique en
         cantidadNotas */
         Desde contador = 1 hasta cantidadNotas paso 1
                  Leer nota       /* Leemos la primera nota*/
                  Mientras nota < 0 ó nota > 10 hacer
                      Escribir (“Nota errónea.Valor mínimo 0 y máximo 10. Vuelva
                           Leer nota
                  Fin-Mientras
                        /* Después de cada nota correcta leída se acumula a la suma to
                  sumaTotal ← sumaTotal + nota
         Fin-Desde
               /* una vez leídas todas las notas correctamente, calculamos la media */
         media ← sumatotal / cantidadNotas
               /* Finalmente escribimos adecuadamente el resultado */
         Escribir (“La media de las ” , cantidadNotas, “  notas leídas es ” , mec
Fin
```



Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

9.4. Lectura de 100 números, indicando cantidad de positivos, negativos y nulos, calculando la suma de cada tipo

Unidad Didáctica IV

Lectura de 100 números, indicando cantidad de positivos, negativos y nulos, calculando la suma de cada tipo

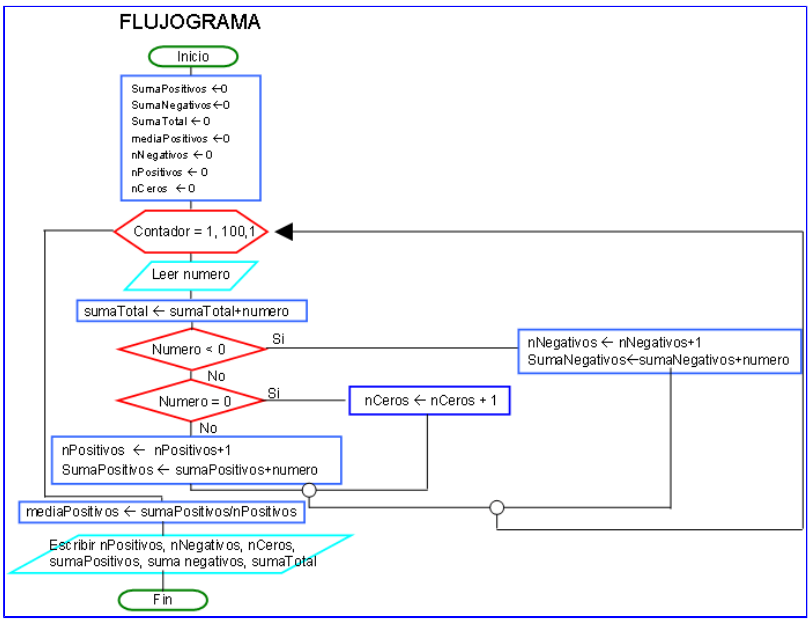
En este ejercicio estamos ante un programa que debe repetir un conjunto de pasos un número fijo de veces, por lo que es adecuado para una estructura tipo FOR.

Queremos leer una serie de 100 números enteros desde teclado, indicando cuántos de ellos son positivos, cuántos negativos, y cuántos cero. También se pide calcular la suma de los negativos, la suma de los positivos y su media, y la suma de los 100. Se desea mostrar convenientemente los valores calculados.

PSEUDOCÓDIGO:

```
Algoritmo SumaPositivosNegativosNulos

Var
   Contador, numero, nPositivos, nNegativos: entero
   nCeros, sumaPositivos, sumaNegativos, sumaTotal : entero
   mediaPositivos : real         /* La media la vamos a calcular con decimales */

Inicio

/* Inicializamos todas las variables acumuladoras para asegurarnos de que valen cero */
         sumaPositivos ← 0
         sumaNegativos ← 0
         sumaTotal ← 0
         mediaPositivos ← 0
         nNegativos ← 0
         nPositivos ← 0
         nCeros ← 0

/* Repetimos un bucle 100 veces, leyendo en cada iteración un número y clasificándolo y
         actualizando los acumuladores que correspondan */
         Desde contador = 1 hasta 100 paso 1
                  Leer numero

/* Acumulamos el valor del número, es decir, a la suma total le sumamos el numero que
acabamos de leer */
                  sumaTotal ← sumaTotal + numero

/* Comprobamos si el número es negativo. Si lo es contamos un positivo más y  lo sumamo
 la suma de los positivos.*/
                  Si numero < 0 entonces
                           nNegativos ← nNegativos + 1
                           sumaNegativos ← sumaNegativos + numero

/* Si no es negativo, comprobamos si es cero. Si lo es contamos un cero más.*/
                  Si no
                           Si numero = 0 entonces
                                    nCeros ← nCeros + 1

/* Si no es negativo ni es cero, tiene que ser positivo. Sumamos un positivo más y
acumulamos el número a la suma de positivos */
                           Si no
                                    nPositivos ← nPositivos + 1
```

```
                                        sumaPositivos ← sumaPositivos + numero
                                Fin-Si
                        Fin-Si
                Fin-Desde

        /* Una vez leídos, clasificados y acumulados los 100 número, calculamos la media de los
         positivos encontrados */

                mediaPositivos ← sumaPositivos / nPositivos

        /* Y finalmente escribimos los resultados */

                Escribir ( “En total se han leído ” , nPositivos , “ números positivos, ’
                     negativos ,  “números negativos y “ , nCeros , “ceros”)
                 Escribir (“Los positivos suman ” , sumaPositivos , “ y su media es ” , n
                 Escribir (“Los negativos suman ” , sumaNegativos)
                 Escribir (“La suma de los 100 números leídos es ” , sumaTotal)
        Fin
```



Pulsa en la imagen para verla ampliada

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

9.5. Leer números y contar pares e impares

Unidad Didáctica IV

Leer números positivos y contar: cuántos son pares y cuántos impares

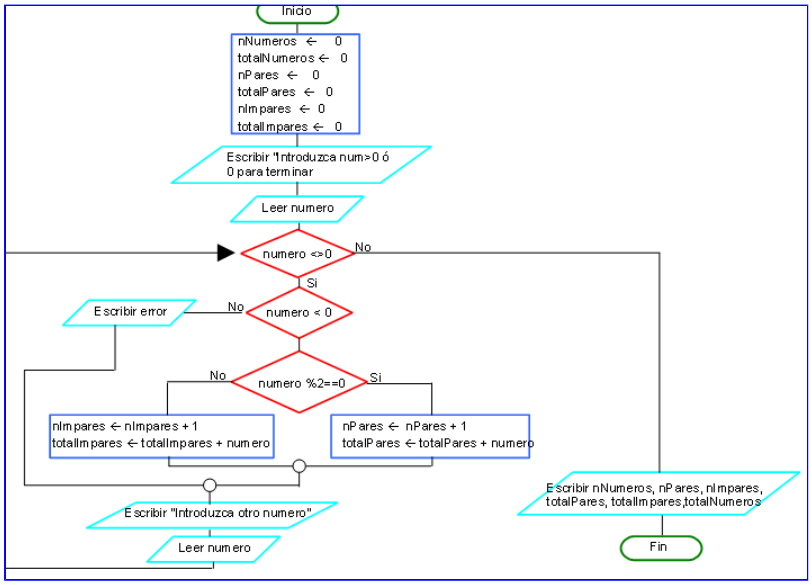
El principal problema de este ejercicio es que no sabemos cuántos números vamos a recibir y además que es preciso distinguir entre pares e impares (para ello debe comparar el resto de dividir el número por 2). Por último es necesario utilizar una variable para contar pares y otra para contar impares.

Queremos leer números positivos para ver si son pares o impares, contando cuántos se leen de cada tipo y calculando la suma de cada tipo y la suma del total de números positivos leídos. También contaremos cuántos números positivos hemos leído en total. Los números negativos serán rechazados, y estaremos leyendo números hasta que se lea un cero. El valor cero lo usaremos como "***centinela***" para saber cuándo tenemos que salir del bucle que nos lee cada número.

PSEUDOCÓDIGO:

```
Algoritmo numerosParesImpares

Var
        numero, nNumeros, totalNumeros, nPares, totalPares, nImpares, totalImpar
Inicio
/* Inicializamos las variables a cero. Para numero no es necesario ya que la prir
lo usamos es justamente para darle un valor leído desde teclado.*/
        nNumeros ← 0
        totalNumeros ← 0
        nPares ← 0
        totalPares ← 0
        nImpares ← 0
        totalImpares ← 0
        /* Leemos el primer número antes de entrar en el bucle.*/
        Escribir ("Introduzca un número positivo o 0 para terminar")
        Leer numero

/* Mientras el nnúmero sea distinto de cero, volvemos a hacer
otra iteración para leer otro nuevo número. La primera vez,
si el número que se ha leído antes del bucle es cero, habremos
terminado sin leer ningún número. Si es distinto de cero, haremos
una iteración en la que procesamos ese número leído, y al final
leemos otro número para la siguiente iteración.*/
        Mientras numero <> 0 hacer
/* Comprobamos si el número leído es negativo. En caso afirmativo,
lo descartamos, y saltamos a la sentencia de lectura del siguiente
número, para volver a leerlo.*/
                Si numero < 0 entonces
                        Escribir ("ERROR: Debe introducir un número positivo o
Vuelva a intentarlo")
/* Si no es negativo, lo procesamos con normalidad, ya que es un nnúmero
 positivo. No puede ser un cero porque si la última lectura fue de un cero,
al comprobar la condición del bucle, sería falsa y no habríamos entrado en
él. Si estamos en este punto, es porque hemos entrado, luego no era un cero */
                Si no
                        totalNumeros ← totalNumeros + numero
                        nNumeros ← nNumeros + 1

/* Si el resto de dividir el numero entre 2 (operador módulo) es cero,
quiere decir que se trata de un número par. En caso contrario, será impar.*/
                        Si (numero mod 2) = 0 entonces
```

```
                                        nPares ← npares + 1
                                        totalPares ← totalPares + numero
                                Si no
                                        nImpares ← nImpares + 1
                                        totalImpares ← totalImpares + numero
                                Fin-Si
                        Fin-Si
                        /* Lectura del número para la siguiente iteración */
                        Escribir ("Introduzca un número positivo o 0 para terminar")
                        Leer numero
                Fin-Mientras
                /* Escritura de los resultados */
                Escribir("Se han leído " , nNumeros, "número positivos, de los que " ,
                        nPares , "son pares y ", nImpares , " son impares")
                Escribir ("La suma de los pares es ",totalPares," y la suma de los impar
                        totalImpares ,". La suma total es  " ,totalNumeros)
Fin
```



Pulse en la imagen para verla ampliada

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

9.6. Cálculo del sueldo neto, en función de las horas trabajadas y de los impuestos a deducir

Unidad Didáctica IV

Cálculo del sueldo neto, en función de las horas trabajadas y de los impuestos a deducir

La dificultad de este ejercicio radica en que utiliza un elevado número de variables y fórmulas que es necesario usar adecuadamente.

Queremos hacer un algoritmo para calcular el sueldo neto semanal de un trabajador y la deducción por impuestos a aplicarle en función de las horas trabajadas en esa semana. Se supone que se introducen por teclado el número de horas trabajadas en la semana junto al nombre del trabajador. Las horas se pagan a tarifa normal hasta las primeras 40 horas, y como tarifa extra las que pasen de esa cantidad. La tarifa normal es de 7,5 € la hora. La tarifa extra es un 50% más cara que la tarifa normal. Una vez calculado el sueldo bruto se calculan los impuestos a deducir, teniendo en cuenta lo siguiente:

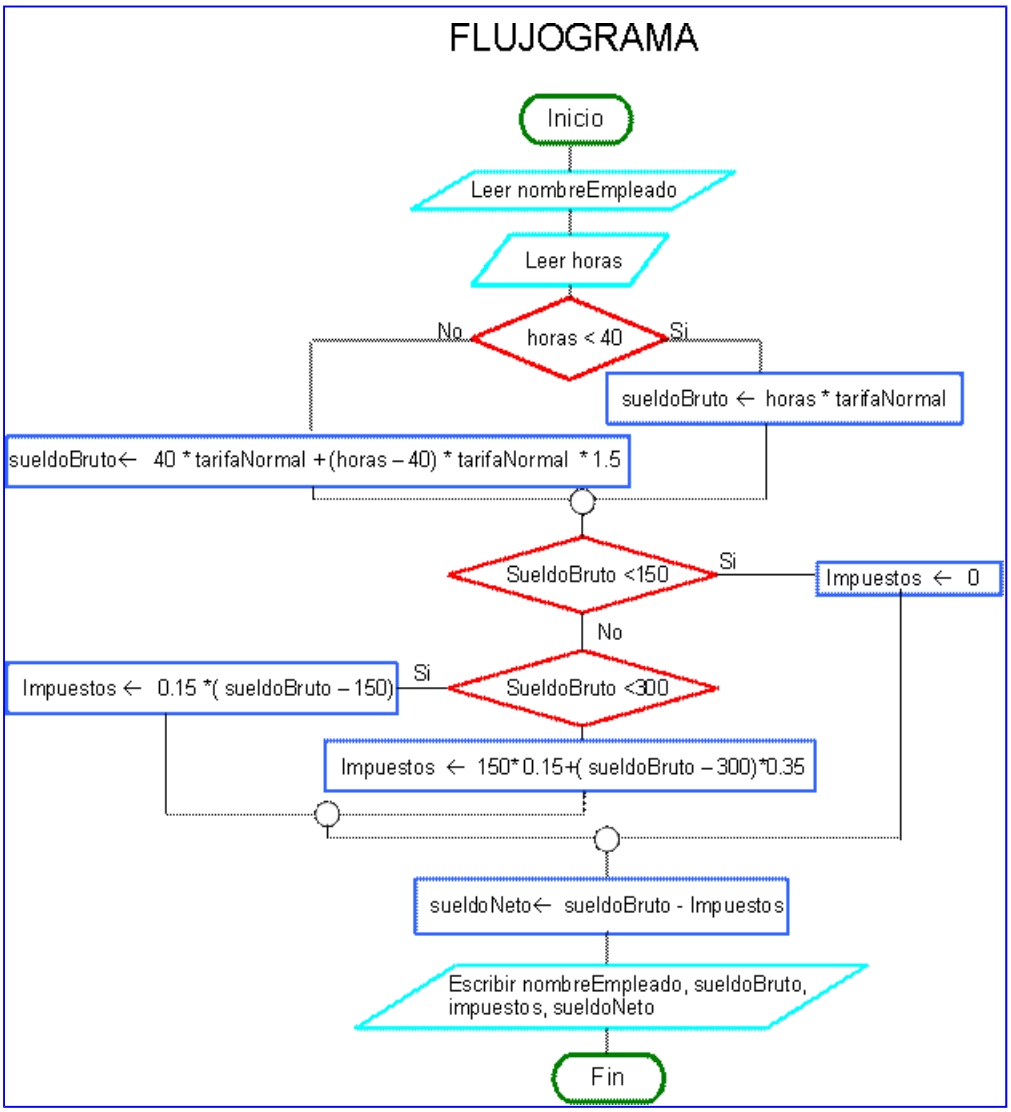
- Los primeros 150 € semanales no pagan impuestos.
- Los siguientes 150 € pagan una tasa de impuestos del 15%
- El resto de salario paga una tasa de impuestos del 35%

Se necesita escribir como salida el nombre del trabajador, su sueldo bruto, los impuestos a pagar y el sueldo neto.

PSEUDOCÓDIGO:

```
Algoritmo CalculoDelSueldo

Const
        tarifaNormal = 7.5
Var
        nombreEmpleado : Cadena
        horas : entero
        sueldoBruto , sueldoNeto, impuestos : real
Inicio
        Leer nombreEmpleado
        Leer horas


/* Las primeras 40 horas se pagan a tarifa normal.*/

        Si horas <40 entonces
                sueldoBruto ← horas * tarifaNormal


/* Las primeras 40 horas se pagan a tarifa normal y el resto a 1,5 veces la tarifa norm
        Si no
                sueldoBruto ← 40 * tarifaNormal + (horas -40) * tarifaNormal *
        Fin-Si


/* Los primeros 150 € no pagan impuestos */
        Si sueldoBruto <= 150 entonces
                impuestos ? 0
        Si no


/* Si el sueldo bruto es mayor de 150 € pero menor o igual que 300, todo lo que pase de
      paga unos impuestos del 15% */
                Si sueldoBruto <= 300 entonces
                        impuestos ← 0.15 * (sueldoBruto – 150)
```

```
/* Si el sueldo es mayor que 300, los primeros 150 € no pagan impuestos, los siguientes
hasta llegar a 300 pagan el 15% y el resto pagan el 35% */
                    Si no
                              Impuestos ← 150 *0.15 + (sueldoBruto -300) * 0.35
                    Fin-Si
          Fin-Si


/* El sueldo neto se calcula restándole los impuestos al sueldo bruto */
          sueldoNeto ← sueldoBruto – impuestos


/* Salida de los resultados */
          Escribir (nombreEmpleado , " →Sueldo Bruto = ", sueldoBruto , ".
           Retención impuestos = " , impuestos , ". SALARIO NETO = ", sueldoNeto ,
Fin
```



Pulsa en la imagen para verla ampliada

9.7. Cálculo del factorial de un número

Unidad Didáctica IV

Cálculo del factorial de un número entero positivo o nulo

El factorial se calcula como un producto de valores que vamos calculando al restar 1 al resultado anterior. Lo más complicado es utilizar el bucle controlado por un contador descendente.

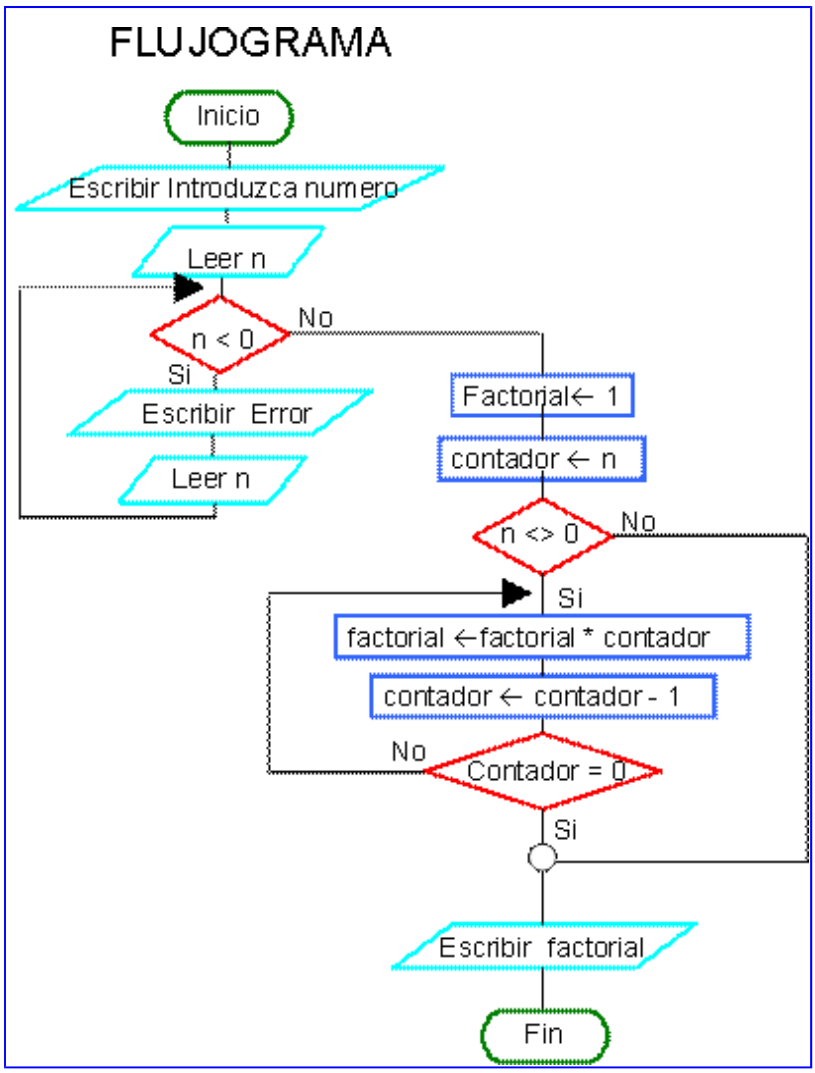
Queremos calcular el factorial de un número entero leído desde teclado. Aunque existe una solución **recursiva** para este problema, no es la que aplicaremos aquí. En su lugar usaremos la solución iterativa. (posteriormente en una unidad dedicada a la recursividad se analizará con detalle el concepto de **recursividad**).

El factorial de un número entero positivo o nulo n se define como:

- n! = 1 , si n vale 0
- n! = n * (n-1) * (n-2) * (n-3) * ··· * 2 * 1 , si n >=1

PSEUDOCÓDIGO:

```
Algoritmo Factorial

Var
        n, factorial, contador : entero
Inicio
     /* Realizamos la primera lectura */
        Escribir ("Introduzca un número positivo o nulo para calcular su factori
        Leer n
     /* Si el número leído es negativo, y mientras que los que sigamos leyendo
lo sigan siendo, será descartado y vuelto a leer. Por tanto, no saldrá del
 bucle hasta haber leído un número mayor o igual a cero.*/
        Mientras n < 0 hacer
                Escribir ("ERROR: sólo se admiten números positivos o nulos par
                                        cálculo del factorial. Vuelva a intr
                Leer n
        Fin-Mientras
     /* Inicializamos factorial a 1 para que al multiplicar la primera vez
tengamos el resultado correcto */
        factorial = 1
     /* contador va a ser usado para ir tomando los valores n, n-1, n-2, ...
que son los factores que hay que ir multiplicando para calcular el
factorial. Por eso lo inicializamos al valor de n, y en cada iteración
del bucle repetir iremos  decrementándolo en una unidad para ir multiplicando
 por un nuevo factor.*/
        contador ← n
     /* Si el número es distinto de cero, calculamos el valor de factorial con el
 ciclo repetir, en el que empezamos multiplicando por contador, inicializado
al valor de n. Si es cero, no tenemos que hacer nada, puesto que su factorial
vale 1, que es justamente el valor de inicializaci&#243;n de la variable
factorial.*/
        Si n <> 0 entonces
                Repetir
                        factorial ← factorial * contador
                        contador ← contador - 1
                Hasta contador = 0
        Fin-Si

        /* Escritura del resultado */
        Escribir (n, "! = " , factorial)
Fin
```



Pulsa en la imagen para verla ampliada

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

9.8. Cálculo de soluciones reales de ecuación de segundo grado

Unidad Didáctica IV

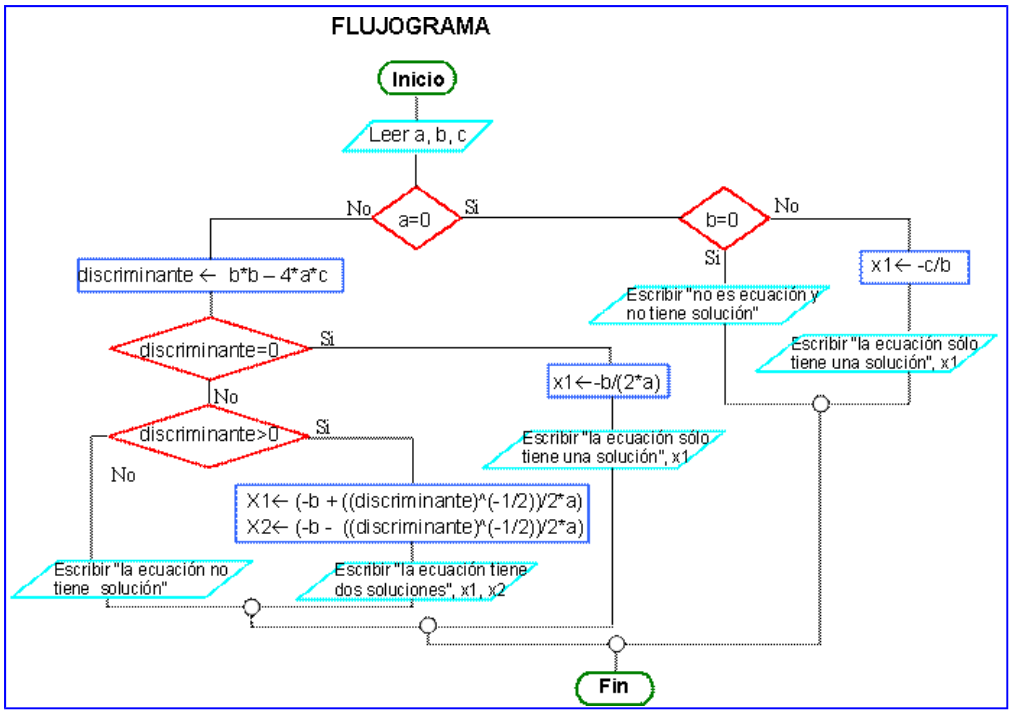
Cálculo de las soluciones reales de una ecuación de segundo grado

Éste es un ejercicio típico de programación, ya que debido al resultado obtenido en el discriminante de la fórmula las soluciones obtenidas pueden variar desde dos raíces reales a no tener ninguna. Presta atención a la forma en que se tratan las fórmulas.

Dada una ecuación de segundo grado con la forma general **$ax^2 + bx + c = 0$**, y leídos desde teclado los valores de sus coeficientes (**a, b, c**), se quiere determinar si tiene solución real o no, y si tiene solución, calcularla y escribirla. La fórmula de la ecuación de segundo grado se supone conocida: **$x = (-b \pm RaizCuadrada(b^2 - 4*a*c)) / (2*a)$**

PSEUDOCÓDIGO:

```
Algoritmo EcuacionSegundoGrado

Var
        a, b, c, discriminante, x1, x2: real
Inicio
        Leer a
        Leer b
        Leer c
        Si a = 0 entonces       /* No es de segundo grado */
                Si b = 0 entonces       /* Ni siquiera es una ecuación */
                        Escribir ("Error: No es una ecuación, y por tanto no t
                Si no       /* De primer grado bx+c=0 */
                        x1 ← -c/b
                        Escribir ("Realmente es una ecuación de primer grado,
                Fin-Si
        Si no       /* De segundo grado */
                discriminante ← b*b - 4*a*c
                Si discriminante = 0 entonces
                        x1 ← -b/(2*a)
                        Escribir ("Sólo tiene una solución, aunque es doble, q
                Si no       /* Discriminante distinto de cero */
                        Si discriminante > 0       /* distinto de cero y además p
                                x1 ← (-b + RaizCuadrada(discriminante))/(2*a)
                                x2 ← (-b - RaizCuadrada(discriminante))/(2*a)
                                Escribir("Tiene dos soluciones simples, que s
                        Si no       /* discriminante distinto de cero, pero negati
que no se puede calcular su raíz cuadrada dentro de los números reales */
Escribir ("No tiene solución real, aunque sí se podría resolver en el conjunto de
                        Fin-Si
                Fin-Si
        Fin-Si
Fin
```



Pulsa en la imagen para verla ampliada

PARA SABER MÁS.
Algunos algoritmos resueltos que pueden ayudarte a entender la resolución de problemas y el diseño de algoritmos. En el siguiente enlace vas a encontrar un resumen del tema de algoritmos, con ejemplos de elaboración y al final unos cuantos ejemplos resueltos.
Teoría de Algoritmos. **[Versión en Caché]**

(Si tienes problemas para acceder a algún enlace, pulsa en "Versión en Caché" para visualizar una copia de esa página web).

Algoritmos: Representación, instrucciones, estructuras básicas de tratamiento, pseudocódigo

10. Apéndice: Ejemplos de programas

Unidad didáctica IV

Apéndice: Ejemplos de programas

Ordenador

En el apartado anterior realmente hemos terminado la unidad 4 propiamente dicha. Llevamos ya pues 4 unidades, estudiando la programación desde un punto de vista todavía bastante teórico, y en esta última, hemos empezado a hacer prácticas de resolución de problemas mediante algoritmos, pero todavía, de programar, nada de nada. Además, con los algoritmos, que se hacen "a papel y lápiz" seguramente en más de un caso habrás sentido que tienes que hacer un "acto de fe" para creerte que realmente se ha resuelto el problema que se pretendía, porque ejecutarlos y ver que funcionan no es posible, más que si nosotros hacemos mentalmente de "procesadores" del algoritmo, algo que requiere un cierto esfuerzo mental, y que seguramente no te habrá resultado fácil.

A estas alturas del curso, lo que seguramente te estarás preguntando es si valía la pena matricularse para aprender a hacer esquemas a lápiz y papel. Seguramente tú pensabas que aquí te íbamos a enseñar a programar de verdad, con el ordenador y con un lenguaje. ¿Para cuando vamos a empezar de verdad a programar en Java y a hacer cosas que realmente resuelvan problemas con un ordenador?

Ordenador

Ante esas preguntas, que sabemos que es posible que te plantees tú, y algún que otro compañero más, lo primero que tengo que pedirte es un poco de paciencia. No dudes de que llegará el momento. Pero la programación no es una disciplina fácil. Requiere entrenamiento, y adaptar nuestros esquemas mentales a la forma de trabajar que tiene un ordenador, para saber plantear las soluciones justamente de esa manera, y eso es lo que vamos consiguiendo con estas primeras unidades. Aún nos queda algo de teoría por ver antes de meternos de lleno con el estudio del lenguaje, pero cada vez menos. Y aunque el trabajo por ahora sea arduo, tendrá su recompensa al empezar a programar de verdad. Todo el trabajo de estas unidades es justamente lo que te permitirá avanzar rápidamente en las próximas.

Pero como a veces la promesa que hace un profesor no siempre es garantía de que se cumplan las expectativas del alumno y a sabiendas de que puede incluso que te estés planteando qué tipo de programas acabarás sabiendo hacer cuando finalices este módulo, hemos pensado oportuno mostrarte el aspecto de varios programas hechos por nuestros alumnos en la modalidad presencial, para que te hagas una idea.

Hemos elegido 4 ejemplos, aunque podríamos haber elegido muchos más, con los programas que realizaron como proyecto final del módulo profesional, en el que aplicar todo lo aprendido en el curso. Entre ellos hay proyectos muy buenos y otros más normales. Evidentemente, no es obligatorio sacar un 10 en el proyecto para aprobar. Pero tienen en común que ninguno de estos alumnos sabía programar previamente. Sólo uno de ellos había hecho ya alguna pequeña aplicación por su cuenta, aprendiendo de forma autodidacta, y más como afición que otra cosa, pero nada más.

Con estos ejemplos queremos conseguir varias cosas:

- Mostrarte lo que queremos enseñarte a hacer en este módulo profesional.
- Mostrarte lo que **cualquier alumno** puede y debe saber hacer al final de este

módulo profesional, aunque no tuviera ningún conocimiento previo de programación. Que un profesor te ponga un magnífico ejemplo, seguramente no te diría mucho sobre lo que un alumno puede hacer. Pero esto son ejemplos reales, hechos por alumnos reales, exactamente como tú y tus compañeros.

- Animarte a seguir esforzándote. Casi todos nuestros alumnos han tenido más o menos dificultades en los comienzos de este módulo, pero con trabajo y constancia, lo han superado, consiguiendo resultados más que satisfactorios, tanto para nosotros como profesores, como para ellos como alumnos.

- Conseguir que te hagas una idea de hacia donde vamos, cosa que a veces no está clara en los comienzos de cualquier tipo de estudios.

Sólo te mostramos el aspecto de estos ejemplos, sin proporcionar el código fuente, por varios motivos.

- Para instalarlos y ejecutarlos, necesitas conocimientos de Java, y de bases de datos que por ahora no tienes en absoluto.(O no tienes porqué tenerlos, ya que no se han visto aún en el curso, y muchos de ellos no se verán hasta las últimas unidades).

- Es una forma fácil de mostrar resaltados los aspectos de la aplicación en los que queremos que te fijes especialmente, con la seguridad de que todos vais a poder verlos correctamente.

- Por otro lado, el esfuerzo empleado por nuestros alumnos para desarrollar estos ejemplos es el mismo que queremos que hagáis todos vosotros, y no nos parece oportuno que el código de sus aplicaciones esté disponible para todo el mundo.

Por último decir que en las presentaciones hemos incluido el nombre y el curso de cada uno de estos alumnos, como reconocimiento a su trabajo, y en agradecimiento a que hayan accedido de forma altruista a que podamos usar sus aplicaciones como ejemplos en este curso, porque ellos también tuvieron en su momento la oportunidad de ver los ejemplos de sus compañeros de años anteriores, y también les sirvió de motivación y de estímulo. Esperamos que a ti te produzca el mismo efecto, porque esperamos que tú lo hagas como mínimo igual de bien que ellos.

Apéndice: Ejemplos de programas

10.1. Primer ejemplo: Gestión de Videoclub

Unidad didáctica IV

Primer ejemplo: Gestión de Videoclub

Este ejemplo es una aplicación de gestión bastante completa de los alquileres de películas y ventas de artículos de un videoclub llamado **Videoclub Infierno**, realizada por el alumno de 1º CFGS de Desarrollo de Aplicaciones Informáticas del IES Aguadulce, **Francisco Molina Bautista**, como proyecto final para el módulo de Programación en Lenguajes Estructurados durante el curso 2004-2005

DEMO: Vea un ejemplo de la aplicación. Parte I, Parte II, Parte III, Parte IV .

Apéndice: Ejemplos de programas

10.2. Segundo ejemplo: Gestión de Herramientas y Obras

Unidad didáctica IV

Segundo ejemplo: Gestión de Herramientas y Obras

Este ejemplo es una aplicación de gestión bastante completa de las herramientas asignadas a las distintas **Obras de una Constructora**, de nombre **Marseg**, realizada por el alumno de 1º CFGS de Desarrollo de Aplicaciones Informáticas del IES Aguadulce, **Francisco Sola Hidalgo**, como proyecto final para el módulo de Programación en Lenguajes Estructurados durante el curso 2004-2005

DEMO: Vea un ejemplo de la aplicación.

Apéndice: Ejemplos de programas

10.3. Tercer ejemplo: Gestión de ventas y alquileres de una Inmobiliaria.

Unidad didáctica IV

Tercer ejemplo: Gestión de ventas y alquileres de una Inmobiliaria.

Este ejemplo es una aplicación de gestión bastante completa de los inmuebles, los clientes, los alquileres y las ventas de una **inmobiliaria** de nombre **InmoSoft**, realizada por el alumno de 1º CFGS de Desarrollo de Aplicaciones Informáticas del IES Aguadulce, **Carlos Carmona Alcántara**, como proyecto final para el módulo de Programación en Lenguajes Estructurados durante el curso 2004-2005

DEMO: Vea un ejemplo de la aplicación.

Apéndice: Ejemplos de programas

10.4. Cuarto ejemplo: Gestión de pedidos de una Pizzería

Unidad didáctica IV

Cuarto ejemplo: Gestión de pedidos de una Pizzería

Este ejemplo es una aplicación de gestión de los pedidos, los clientes y la facturación de una **pizzería** llamada **Venecia**, realizada por el alumno de 1º CFGS de Desarrollo de Aplicaciones Informáticas del IES Aguadulce, **José Antonio Ruíz Santiago**, como proyecto final para el módulo de Programación en Lenguajes Estructurados durante el curso 2004-2005

DEMO: Vea un ejemplo de la aplicación.

Apéndice: Ejemplos de programas