ESCUELA DE PREPARACIÓN DE OPOSITORES
E. P. O.
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

INF23-SAI25

Diseño de algoritmos. Técnicas descriptivas.

Esquema.

1 Introducción.

Un computador es capaz de realizar determinadas acciones sencillas, tales como sumar, restar o transferir datos. Estas acciones son de por si útiles; sin embargo, los problemas que normalmente interesa resolver son más complejos. Para solucionar un

problema real, es necesario encontrar un método de resolución del problema y, posteriormente, descomponerlo en acciones sencillas, que el computador sea capaz de realizar.

En este tema nos centraremos en el concepto de algoritmo, introduciendo distintas representaciones para éstos. A continuación describiremos las acciones fundamentales en lenguajes de programación procedimentales, y concluiremos nuestro estudio presentando algunas de las técnicas más utilizadas para el diseño de algoritmos.

2 Concepto de algoritmo.

No todos los métodos de solución de un problema son susceptibles de ser utilizados por un computador. Para que un procedimiento pueda ser implantado en un computador, o en otra máquina capaz de interpretar instrucciones, debe cumplir determinados requisitos:

- El procedimiento debe estar compuesto de acciones bien definidas, esto es, no ambiguas. El significado de cada acción debe ser único, en el contexto en que aparece.
- El procedimiento debe estar formado por una secuencia finita de operaciones. Además, debe quedar perfectamente definido el orden en que se van a realizar las instrucciones.
- Por último, el procedimiento debe acabar en un tiempo finito. Un procedimiento que puede no acabar nunca no es útil para resolver un problema.

Un procedimiento o método de solución, para resolver un problema, que cumpla estos requisitos se dice que es un algoritmo que resuelve ese problema.

Se puede dar la siguiente definición de algoritmo: Un algoritmo es un procedimiento no ambiguo que resuelve un problema. Un procedimiento es una secuencia de operaciones bien definidas, cada una de las cuales requiere una cantidad finita de memoria y se realiza en un tiempo finito.

Supongamos que se quiere explicar a alguien lo que debe hacer para determinar si un número es par. Se le puede decir: Si el número se puede obtener sumando doses, es par. Si para construirlo hay que sumar a uno cualquier secuencia de doses es impar.

Se entiende perfectamente lo que se quiere decir. No obstante, este enunciado no es un algoritmo, pues no constituye una secuencia de operaciones. Además, presenta ambigüedad. Este mismo enunciado se podría expresar de una forma más precisa, del siguiente modo:

```
1. Leer N
2. Si N=2 entonces Escribe ("es par")
3. Si N=1 entonces Escribe ("es impar")
4. N=N-2
5. Si N>0 ir a 2
6. Fin
```

Ahora se ha definido una secuencia de operaciones. Para que esta secuencia sea un algoritmo, las operaciones que aparecen en ella han de ser no ambiguas, y para cualquier dato de entrada el proceso debe acabar en un tiempo finito.

Es suficiente una breve inspección del procedimiento para comprobar que éste acaba después de (N+1)/2 iteraciones. Por tanto, este procedimiento es un algoritmo.

Para resolver un mismo problema, se pueden definir infinidad de algoritmos. Es posible realizar comparaciones entre algoritmos que resuelvan un mismo problema. Normalmente interesa, no sólo encontrar un algoritmo, sino que éste sea suficientemente "bueno". La bondad de un algoritmo puede medirse por dos factores:

- El tiempo que se necesita para ejecutarlo. Para comparar dos algoritmos no es necesario conocer el tiempo real que invertirá el computador, basta con saber el número de instrucciones de cada tipo necesarias para resolver el problema.
- Los recursos que se necesitan para implementar el algoritmo. Concretamente, en el caso de que el algoritmo lo deba ejecutar un computador, se usan fundamentalmente los siguientes recursos: memoria principal para almacenar los datos y las instrucciones, y memoria masiva para almacenar datos auxiliares.

No hay ningún procedimiento riguroso que permita construir un algoritmo que resuelva un problema dado. El diseño de algoritmos, como toda tarea creativa, es una tarea compleja, en la que se pueden seguir pocas normas, teniendo por tanto gran importancia la imaginación y experiencia de la persona que lo realiza. Tan sólo se pueden dar normas generales y herramientas que pueden ayudar a confeccionar el algoritmo.

El algoritmo definido en el ejemplo anterior necesita (N+1)/2 iteraciones para determinar si el número N es par. El tiempo que tarda el algoritmo en resolver el problema aumentará linealmente con la magnitud del número. El problema planteado se puede resolver de forma más simple, observando que los números pares son iguales al doble de la parte entera de su mitad. El siguiente algoritmo se basa en esta idea.

```
Leer N
M=2*int(N/2)
Si M=N entonces escribe "es par"
Si no escribe "es impar"
```

Este algoritmo resuelve el problema en un tiempo constante, es decir, el tiempo de resolución no depende de la magnitud del número introducido.

Por tanto es necesario, una vez diseñado un primer algoritmo, realizar una evaluación del mismo. Si se decide que éste no es eficiente, será necesario o bien diseñar uno nuevo, o bien optimizar el original. Optimizar un algoritmo consiste en introducir modificaciones en él tendentes a disminuir el tiempo que necesita para resolver el problema, o a reducir los recursos que utiliza. En muchos casos ambas acciones se contraponen, o sea, que una disminución del tiempo implica utilizar más variables. La importancia de cada uno de estos factores dependerá de la naturaleza del problema y de los medios de que se disponga.

Una vez ideado el algoritmo, será necesario crear un programa que sea transcripción del algoritmo en algún lenguaje de programación. Las etapas a seguir en este proceso se analizarán más adelante.

3 Representación de algoritmos.

Hay diferentes métodos para representar los algoritmos. Por supuesto un método es narrar, o enunciar el algoritmo. Para facilitar esta descripción es frecuente utilizar un lenguaje de descripción de algoritmos o "pseudocódigo". Existen otros procedimientos de representación que utilizan gráficos o diagramas. Entre estos últimos caben destacar

los organigramas o diagramas de flujo y los diagramas de Chapin o diagramas de Nassi-Schneiderman (abreviadamente diagramas N-S).

3.1 Pseudocódigo.

No hay reglas fijas para la representación narrativa de algoritmos. No obstante, para describir algoritmos está muy extendido el uso de las estructuras de control del lenguaje Pascal. En este caso, como el objetivo no es escribir un programa para ser ejecutado por un computador, no hay reglas sintácticas estrictas, el interés se centra en la secuencia de instrucciones. Este tipo de descripción se denomina pseudocódigo. La utilización de pseudocódigo presenta las ventajas de: ser más compacto que un organigrama, ser más fácil de escribir, y ser más fácil de transcribir a un lenguaje de programación.

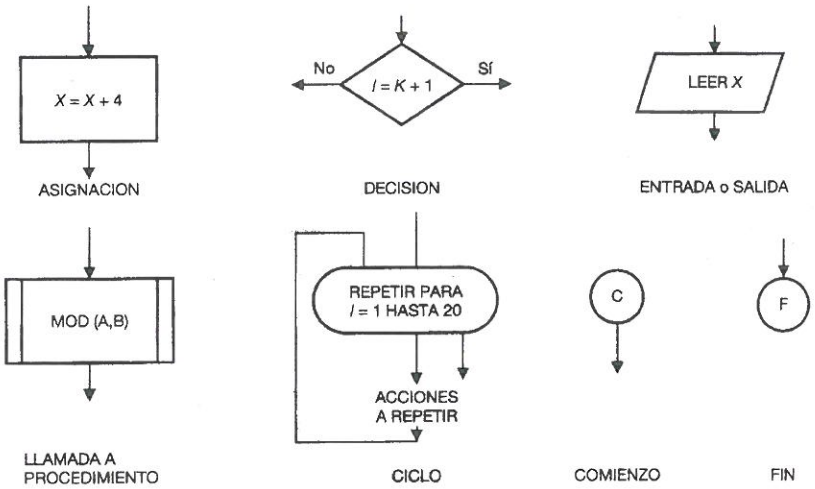
Un algoritmo escrito en pseudocódigo presenta una estructura semejante a la de un programa. El algoritmo del primer ejemplo no está escrito en pseudocódigo, aunque sí en forma narrativa. Dicho algoritmo se puede escribir en pseudocódigo, tal como se muestra a continuación.

```
Leer N
mientras N>2 repetir
   N=N-2
si N=2 entonces escribe "es par"
si no escribe "es impar"
fin
```

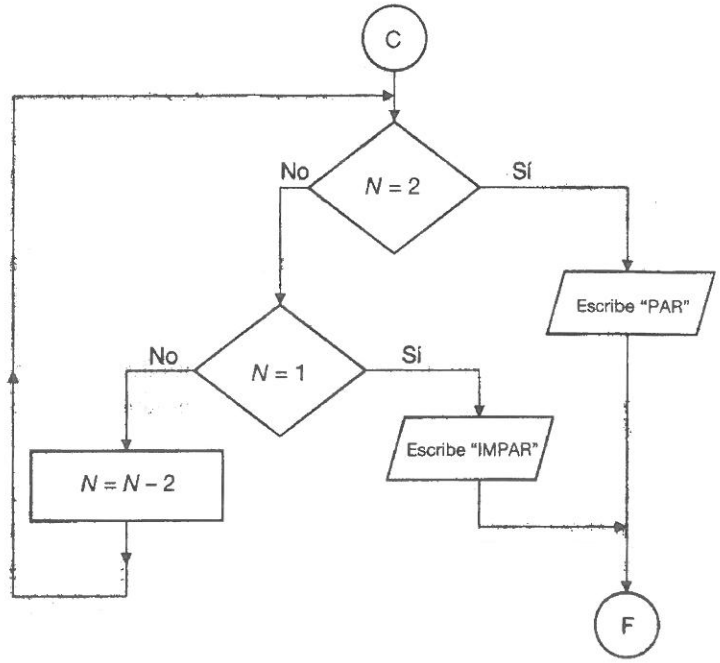
3.2 Organigramas.

Los organigramas son herramientas gráficas para representar algoritmos. Un organigrama está compuesto por una serie de símbolos unidos por flechas. Los símbolos representan acciones, y las flechas el orden de realización de las acciones. Cada símbolo, por tanto, tendrá al menos una flecha que conduzca a él y una flecha que parta de él.

Cada símbolo usado representa una acción distinta. Hay símbolos específicos para representar las siguientes acciones: *asignación*, *lectura/escritura*, *llamada a subrutina*, *ciclo*, *decisión*, *comienzo*, y *fin*. En la figura se muestran los símbolos habitualmente utilizados en la confección de organigramas.



El organigrama del ejemplo anterior sería entonces:

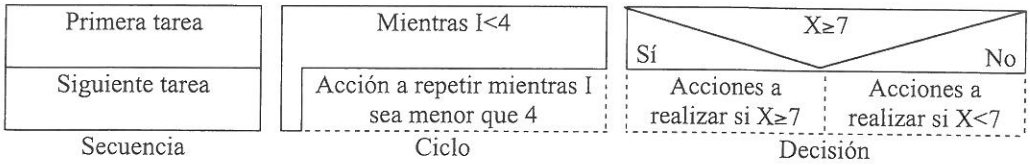


3.3 Diagramas de Nassi-Schneiderman.

Los organigramas descritos anteriormente se pueden utilizar para representar cualquier algoritmo. No obstante, los diagramas de Nassi-Schneiderman (o diagramas de Chapin) tienen la ventaja de adecuarse a las técnicas de programación estructurada. Los diagramas N-S no utilizan flechas para indicar el flujo de control. Además estos organigramas tienen otras ventajas respecto a los organigramas clásicos, como son:

- Se leen de arriba a abajo, al igual que un programa estructurado. El organigrama queda, por tanto, más claro.
- Permiten el uso de técnicas de diseño descendente.
- Favorecen la partición de los programas en módulos pequeños.
- Resaltan más las partes generales, quedando los detalles tanto más pequeños cuanto más específicos.

Un diagrama N-S es un dibujo contenido en un rectángulo. Dentro del rectángulo se incluyen una serie de símbolos adyacentes. Los símbolos usados corresponden a las formas de transferencia de control: ciclo, decisión y transferencia secuencial. Los símbolos usados se muestran en la figura.

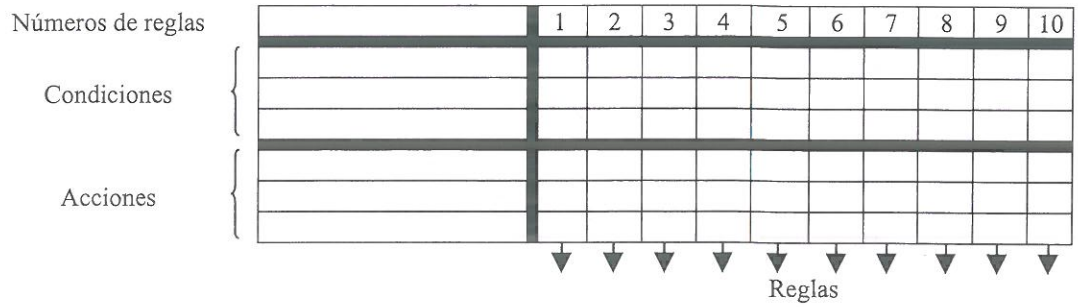


3.4 Notaciones tabulares de diseño.

En muchas aplicaciones de software, puede ser necesario que un módulo evalúe una compleja combinación de condiciones y, de acuerdo con ellas, seleccione la acción

apropiada. Las tablas de decisión constituyen una notación que traduce las acciones y las condiciones a una forma tabular. Es difícil que se interprete mal la tabla, e incluso puede usarse como entrada interpretable por la máquina para un algoritmo manejado por tabla. En un amplio tratamiento de esta herramienta de diseño, Ned Chapin establece: "Algunas antiguas herramientas y técnicas del software se entremezclan bien con las nuevas técnicas y herramientas de la ingeniería del software. Las tablas de decisión son un ejemplo excelente. Las tablas de decisión surgieron casi una década antes que la ingeniería del software, pero le van tan bien a la ingeniería del software que podrían haber sido diseñadas para dicho propósito".

La siguiente figura muestra la organización de una tabla de decisión.



Refiriéndonos a la figura, las líneas más gruesas dividen la tabla en cuatro cuadrantes. El cuadrante superior izquierdo contiene una lista con todas las condiciones. El cuadrante inferior izquierdo contiene una lista de todas las acciones que se pueden producir basándose en combinaciones de las condiciones. Los cuadrantes de la derecha forman una matriz que indica las combinaciones de las condiciones y las correspondientes acciones que se han de producir para cada combinación específica. Por tanto, cada columna de la matriz puede interpretarse como una regla de procesamiento.

Para desarrollar una tabla de decisión se aplican los siguientes pasos:

1. Listar todas las acciones que puedan asociarse con un módulo.
2. Listar todas las condiciones (o decisiones hechas) durante la ejecución del procedimiento.
3. Asociar conjuntos específicos de condiciones con acciones específicas, eliminando las combinaciones de condiciones imposibles; alternativamente, desarrollar cada posible permutación de condiciones.
4. Definir reglas indicando qué acciones ocurren para un conjunto de condiciones.

Para ilustrar el uso de una tabla de decisión, consideremos el siguiente extracto de una narrativa de procesamiento para un sistema de facturación de un servicio público: "... si la cuenta del cliente se factura usando un método de tarificación fijo, se establece una carga mensual mínima para consumos menores de 100 KWh (kilowatios hora). En los demás casos, la facturación por computadora aplica la tarifa A. Sin embargo, si la cuenta se factura usando un método de facturación variable, se aplicará la tarifa A a los consumos menores de 100 KWh, con un consumo adicional facturado de acuerdo a la tarifa B".

La figura ilustra una tabla de decisión que representa el anterior texto.

	1	2	3	4	5
Cuenta de tarifa fija	T	T	F	F	F
Cuenta de tarifa variable	F	F	T	T	F
Consumo < 100 KWh	T	F	T	F	
Consumo ≥ 100 KWh	F	T	F	T	
Cargo mensual mínimo	X				
Esquema de tarificación A		X	X		
Esquema de tarificación B				X	
Otro tratamiento					X

Cada una de las cinco reglas indica una de cinco posibles condiciones (por ejemplo, una "V" (verdad) en las cuentas de tarificación fija y variable no tiene sentido en el contexto de este procedimiento). Como regla general, la tabla de decisión puede usarse con efectividad para complementar a otras notaciones de diseño procedimental.

3.5 Comparación de las notaciones de diseño.

Hemos presentado varias notaciones de diseño procedimental. Cualquier comparación debe basarse en la premisa de que cualquier notación para el diseño procedimental, si se usa, correctamente, puede ser una ayuda muy valiosa en el proceso de diseño; recíprocamente, incluso la mejor notación, si se aplica mal, añade poco a la comprensión. Con esto en mente, podemos examinar los criterios que pueden aplicarse para comparar notaciones.

Una notación de diseño debe conducir a una representación procedimental que sea fácil de comprender y revisar. Además, la notación debe facilitar la "codificación", de forma que el código se obtenga de hecho como un producto natural del diseño. Finalmente, la representación del diseño debe ser fácilmente mantenible, de forma, que el diseño represente siempre correctamente el programa.

Los siguientes atributos de las notaciones de diseño se han establecido en el contexto de las características generales descritas anteriormente:

- Modularidad. Una notación de diseño debe soportar el desarrollo de software modular (por ejemplo especificación directa de procedimientos y estructuración en bloques) y suministrar una forma de especificación de las interfaces.
- Simplicidad global. Una notación de diseño debe ser relativamente sencilla de aprender, relativamente fácil de usar. y, generalmente fácil de leer.
- Facilidad de edición. El diseño procedimental puede requerir modificaciones durante el paso de diseño, durante la prueba del software y, finalmente, durante la fase de mantenimiento del proceso de ingeniería del software. La facilidad con la que una representación de diseño pueda ser editada, puede ayudar a facilitar cada uno de estos pasos de la ingeniería del software.
- Legible por la máquina. Los entornos de ingeniería del software asistida por computadora están siendo adoptados por la industria. Una notación que pueda ser introducida directamente en un sistema de desarrollo basado en computadora ofrece unos enormes beneficios potenciales.
- Mantenimiento. El mantenimiento del software es la fase más costosa del ciclo de vida del software. El mantenimiento de la configuración del

software casi siempre significa mantenimiento de la representación del diseño procedimental.

- Exigencia de estructura. Ya se han visto las ventajas de un método de diseño que utilice los conceptos de la programación estructurada. Una notación de diseño que refuerce el uso sólo de construcciones estructuradas promueve la práctica de un buen diseño.
- Procesamiento automático. Un diseño detallado contiene información que puede ser procesada para dar al diseñador nuevos o mejores conocimientos respecto a la corrección y la calidad de un diseño. Dichos conocimientos se pueden ampliar con informes obtenidos mediante un procesador automático.
- Representación de los datos. La habilidad para representar datos locales, y globales es un elemento esencial en el diseñó detallado. Idealmente, una notación de diseño debe representar directamente dichos datos.
- Verificación lógica. La verificación automática de la lógica de un diseño es un objetivo principal durante la prueba del software. Una notación que refuerce la posibilidad de verificar la lógica, mejora bastante la idoneidad de la prueba.
- Disposición para la codificación. El paso de ingeniería del software que sigue al diseño procedimental es la codificación. Una notación que se convierta fácilmente a código fuente reduce el trabajo y los errores.

Una pregunta natural que surge en cualquier tratamiento sobre las notaciones de diseño es: "¿qué notación es realmente la mejor, dados los atributos mencionados anteriormente?". La respuesta a esta pregunta definitivamente es subjetiva y está sujeta a debate. Sin embargo, parece que un lenguaje de diseño de programas ofrece la mejor combinación de características. El lenguaje de programación puede insertarse directamente en los listados fuente, mejorando de esta forma la documentación y haciendo menos difícil el mantenimiento del diseño. La edición puede conseguirse con un editor de texto o un sistema de procesamiento de texto; ya existen procesadores automáticos y buenas expectativas para la "generación automática de código".

Sin embargo, de esto no se ha de deducir que las otras notaciones de diseño sean necesariamente inferiores al lenguaje de diseño de programas o no "sean buenas" para atributos específicos. La naturaleza gráfica de los diagramas de flujo y los diagramas de Nassi-Schneiderman proporcionan una perspectiva general del flujo de control que satisface a muchos diseñadores. El contenido tabular preciso de las tablas de decisión es excelente para aplicaciones conducidas por tablas.

4 Acciones y estructuras de control.

A continuación se tratará de las distintas acciones que pueden utilizarse en un algoritmo o en un programa y de las estructuras que se pueden usar para controlar el orden de realización de éstas.

Entre las acciones comentaremos las asignaciones y las entradas/salidas. De las estructuras de control se analizarán las decisiones y los ciclos. Se introducirá también el concepto de procedimiento (o subrutina). Estas no son las únicas acciones y estructuras de control que pueden aparecer en un lenguaje de programación o en la descripción de un algoritmo, pero son suficientes para resolver cualquier problema.

En un ciclo repetitivo se debe especificar siempre el valor inicial y final que toma el contador del ciclo, y el incremento a aplicar en cada iteración, si éste es distinto a uno.

Otra estructura de control usual es el ciclo condicional, en el que un segmento de programa se repite mientras (ciclo "While") se cumpla una condición o hasta que (ciclo "Until") se deje de cumplir una condición. La condición siempre es una expresión lógica. Este tipo de ciclos, como se verá más adelante, es más general que el anterior. En la siguiente figura se muestran las estructuras de distintos tipos de ciclos.

4.5 Procedimientos.

Un procedimiento o subrutina es un fragmento del programa que realiza una tarea concreta, y recibe un nombre por el que puede ser llamado o activado desde otra parte del programa. Un procedimiento puede tener una serie de variables de comunicación denominadas argumentos, que permiten el paso de información entre el programa y el procedimiento.

El uso de procedimientos evita la duplicación de grupos de sentencias en diferentes partes del programa. Pero sobre todo facilita la construcción y comprensión de los programas, ya que es más fácil diseñar y poner a punto un segmento de programa pequeño, que realiza una función concreta, que uno grande. Sobre este aspecto volveremos más adelante.

5 Recursividad.

La recursividad está basada en la utilización de procedimientos, y en una técnica especial de resolución aplicable a problemas con "estructura recursiva". Supongamos un problema A. Al estudiar el problema A conseguimos dividirlo en dos partes B y C. Si una de estas partes, por ejemplo C, formalmente idéntica a A, el problema es recursivo, y puede resolverse de forma recursiva.

En términos del lenguaje de programación esto supone utilizar un procedimiento, para resolver ,el problema, que se va a llamar a sí mismo, para resolver parte del problema.

En cualquier caso, la solución recursiva no es nunca la única posible, siempre es factible utilizar un ciclo, en lugar del planteamiento recursivo. No obstante, en algunos casos, la solución recursiva es la solución natural del problema, y por tanto la más clara.

Al realizar un procedimiento recursivo se está haciendo, implícitamente, uso de una pila, ya que los datos locales del procedimiento se crean cada vez que éste es llamado. Si al resolver el problema es necesario utilizar pilas, el uso de procedimientos recursivos puede ser conveniente. En caso contrario normalmente es desaconsejable, pues una solución recursiva es más lenta que una solución iterativa.

6 Proceso de creación de un programa.

Una vez explicadas las herramientas que se han de usar, estamos en condiciones de exponer qué se ha de hacer para desarrollar un programa en un lenguaje de alto nivel. La concepción de cualquier programa comienza con el planteamiento del problema que éste ha de resolver. Cuando se ha adquirido una idea clara y precisa del problema, se debe escoger una representación para los diferentes datos que intervienen en él. Hecho esto, se puede pensar en redactar un algoritmo que resuelva el problema. Este algoritmo ha de depurarse y optimizarse. El paso siguiente es redactar un programa según el

4.1 Asignaciones.

Una asignación consiste en la evaluación de una expresión y en el almacenamiento de su valor en una variable. Este tipo de sentencias se ejecutan siempre secuencialmente. Esto es, a continuación de la asignación se realiza siempre la acción siguiente del programa. La asignación siempre implica una transferencia (movimiento) de datos en memoria. Esto es, un movimiento de datos de unas variables a otras. En algunos casos puede incluir, además, la realización de una serie de operaciones aritméticas, lógicas, o de manejo de caracteres u otras estructuras de datos. La variable a la que se le asigna el nuevo valor pierde el que tuviera anteriormente. Esta acción es esencial en la mayoría de los lenguajes de programación (salvo en los lenguajes lógicos y funcionales).

4.2 Entradas/Salidas.

Las operaciones de entrada/salida se utilizan para que el programa intercambie información con un medio externo. En una operación de entrada o lectura, el programa asigna a una variable un valor almacenado en un soporte masivo o generado por un periférico. En una operación de salida o escritura el programa transfiere a un dispositivo de memoria masiva, o a un periférico, el valor de una variable.

En cualquier caso, en la instrucción se debe especificar el dispositivo en el que se escribe, o del que se lee. En algunos casos se omite el nombre del dispositivo, sobrentendiéndose entonces que éste es el terminal o equipo periférico con el que se está accediendo al computador.

Al igual que en las asignaciones, la entrada/salida no conlleva alteración en el orden de ejecución de las acciones siguientes a ella, salvo situaciones excepcionales como la que se indica en el párrafo siguiente.

Si la lectura se realiza de un archivo, se puede llegar a intentar leer más allá del final de éste. En esta situación se dice que se ha producido un error por fin de archivo. En un lenguaje de programación esta situación se puede tratar de dos modos. En algunos lenguajes la sentencia de lectura tiene una salida especial para fin de archivo, es decir, se produce un salto en la ejecución del programa si se encuentra un fin de archivo. La otra opción (usada, por ejemplo, en Pascal) es que con cada archivo del que se lee esté asociado un valor que indica si se ha alcanzado su final. Este valor se actualiza automáticamente después de cada operación. Esto permite que en el programa se puedan tomar distintas acciones según se haya llegado o no al fin del archivo, simplemente comprobando este valor.

4.3 Decisiones.

Las decisiones son acciones de control de flujo. Permiten modificar el orden en que se realizan otras acciones. Una decisión posee siempre un argumento, que normalmente debe ser una expresión lógica. Dependiendo del valor de la expresión lógica se ejecutan las acciones que están en uno u otro camino, a partir de la decisión. Una decisión permite, por tanto, bifurcar en dos caminos el flujo de acciones.

4.4 Ciclos.

Un ciclo es una estructura de control que indica la repetición de un segmento de programa. El ciclo puede ser repetitivo, en cuyo caso el segmento se repite un número fijo de veces. En este tipo de ciclos existe una variable de control del ciclo a la que automáticamente se le asignan valores sucesivos durante la ejecución del ciclo.

algoritmo encontrado, que habrá de ser igualmente depurado. Si hay algún error en el algoritmo, que no fue detectado en el proceso de depuración de éste, posiblemente sea detectado ahora, y habrá que retroceder varios pasos en el proceso. Por último, se debe optimizar y documentar el programa. A continuación se verán con más detalle cada uno de estos pasos.

6.1 Planteamiento del problema.

Antes de enfrentarse con la resolución de un problema es necesario conocer perfectamente éste. Esencialmente se trata de conocer los resultados a obtener, los datos de que se dispone y la relación entre ellos. Esto es, saber qué datos son, cuál es su significado, qué valores pueden tomar y qué relaciones hay entre ellos. Aquí se deben detectar omisiones en los datos de partida o ambigüedades en la especificación de los resultados.

6.2 Representación de los datos.

La estructura que ha de tener el algoritmo que se desea diseñar depende en gran medida de la representación usada para los datos que intervienen en el problema. Si la representación no es la adecuada, el algoritmo no será bueno. Si el problema a resolver es complejo, será necesario establecer un proceso de ajuste entre las estructuras de datos y el algoritmo, retocando las estructuras de datos después de haber ensayado un algoritmo, hasta llegar a una solución aceptable.

6.3 Diseño de un algoritmo.

Esta es, junto con la fase anterior, la parte más delicada del desarrollo de un programa. Ambas etapas están, además, íntimamente relacionadas, tal como se ha explicado antes. Indudablemente la dificultad que se encuentre dependerá de la complejidad del problema, entre otros factores.

Hay un límite de complejidad, por encima del cual es difícil moverse sin ayudas, independientemente de la experiencia que se tenga. La mente humana está acostumbrada a trabajar con objetos que puede imaginarse, y en número reducido. Siempre habrá problemas lo suficientemente complejos como para no poder tenerlos en mente, en conjunto.

Esto, no obstante, nos da una idea de cómo se ha de afrontar un problema complejo. Podemos ir descomponiendo el problema a resolver en problemas cada vez más simples, hasta llegar a un nivel que seamos capaces de resolver directamente. Esta es, en esencia, la idea del diseño descendente, que trataremos a continuación.

6.4 Diseño descendente.

El diseño descendente es una técnica natural de desarrollo de algoritmos, hasta el punto de que empezó a usarse intuitivamente antes de haberse sistematizado su empleo. Diversos autores han abordado su estudio.

El método consiste en comenzar trabajando a nivel abstracto, para ir dividiendo el problema en sus partes naturales. De esta forma, el problema a resolver se descompone en otros más simples. A estos últimos se les aplica el mismo procedimiento hasta llegar a problemas suficientemente pequeños como para ser resueltos directamente.

Hay otra forma, tal vez más clara, de entender este proceso. Partimos de un problema a resolver y de un repertorio de acciones posibles. Pues bien, olvidémonos de las acciones que podemos realizar, y supongamos que tenemos un ordenador mágico que es capaz de realizar cualquier acción hipotética. Puestas así las cosas, la resolución de cualquier problema puede ser fácil. Las acciones hipotéticas usadas pueden ser complejas y no tendrán que coincidir con las que se tengan en un ordenador real. Una vez encontrado un algoritmo, que usa acciones hipotéticas, se deben considerar individualmente cada una de estas acciones y plantearlas como nuevos problemas, encontrando algoritmos que los resuelvan, para lo que se puede volver a pensar en términos del ordenador mágico. Este proceso se repite sucesivamente hasta llegar a una solución factible del problema.

No obstante no hay que engañarse, puede haber pasos en el diseño que sean especialmente complicados. Hay que pensar que cada nuevo nivel generado debe estar más detallado que el anterior. Esto es, siempre se debe avanzar, pareciéndose cada vez más las acciones hipotéticas a las reales.

Usando este método se van generando soluciones del problema a distintos niveles de detalle. En cada nivel la solución podrá ser comprobada y depurada, antes de pasar a resolver los módulos que la forman a un nivel más bajo. Esto presenta la ventaja de facilitar la tarea de comprobación del algoritmo. El algoritmo, resuelto de esta forma, se puede representar por un árbol en que cada nodo es un módulo (un problema o acción hipotética). Cada subárbol dependiente de un nodo se utiliza en la solución de ese nodo. En particular, el nodo raíz es el problema de partida (nivel 0). El algoritmo será correcto si la solución dada a cada nivel es correcta. El proceso descrito para crear un algoritmo se deberá usar para resolver cada nodo del árbol, desde la descripción hasta la depuración. El programa podrá crearse como un todo, teniendo en cuenta al algoritmo completo. Sin embargo, normalmente será más conveniente dividirlo en módulos, coincidentes con las partes naturales del problema encontradas en el momento del diseño. En este último caso, el programa podría construirse de abajo a arriba (de forma ascendente: del nivel n al 0), creando primero procedimientos que resuelvan los módulos de detalle, que una vez comprobados serán usados por otros procedimientos más generales, hasta llegar a la creación del programa, cuya estructura será la de la solución a nivel 0.

Una ventaja del diseño descendente es que al ser los módulos más pequeños son más fáciles de comprender. Otra ventaja es su adaptación a la programación estructurada.

Los diagramas N-S pueden jugar también un gran papel, pues se adaptan perfectamente al diseño descendente. La solución a nivel cero se puede representar por un diagrama. Al realizar la solución a nivel 1 se detallarán algunos de los cuadros del diagrama del nivel anterior. Así, hasta detallar completamente el organigrama. Si algún módulo interesa desarrollarlo como procedimiento se escribirá su diagrama aparte.

6.5 Comprobación y optimización de algoritmos.

Un último paso a realizar cuando se escribe un algoritmo es comprobar que es correcto, y que realiza las operaciones que se desean. Para comprobar un algoritmo se puede efectuar un seguimiento del mismo. El seguimiento consiste en ejecutar manualmente las operaciones descritas en el algoritmo, según el orden especificado en el mismo, anotando el valor que toma cada dato durante el proceso.

Si el algoritmo se ha creado por diseño descendente la comprobación se realizará nivel a nivel. En cualquier caso, se podrá usar el procedimiento de comprobación descrito anteriormente. En algunos casos se utilizan otros procedimientos de comprobación más rigurosos, que permiten no sólo mostrar que el algoritmo funciona en algunos casos, sino demostrar que algunas partes del mismo son correctas. Concretamente se puede tener en cuenta lo siguiente:

- Para comprobar que un ciclo funciona correctamente se puede aplicar una demostración por inducción matemática.
- Si dos partes del algoritmo son completamente independientes se puede probar el funcionamiento de cada una de ellas independientemente.

En la resolución de cualquier problema es necesario usar variables, para almacenar los datos que se han de manejar. Para que el algoritmo sea legible, se deben especificar en él todas las variables usadas, indicando su tipo y contenido.

La optimización del algoritmo normalmente consiste en buscar uno mejor, para sustituirlo total o parcialmente. Para realizarlo, es necesario tener una idea del comportamiento del algoritmo, es decir, realizar una evaluación del mismo. La evaluación es normalmente una tarea compleja. Esencialmente consiste en calcular el número de sentencias que se habrán de ejecutar en la resolución del problema, en el mejor o peor de los casos, y en una situación media. La mayoría de las veces esto no es viable, entre otras cosas por la dificultad para describir un caso de cada tipo, sobre papel. Frecuentemente debemos conformamos con cotas más o menos próximas a los valores reales.

7 Técnicas de diseño de algoritmos.

7.1 Divide y vencerás.

Tal vez la técnica más importante y aplicada para el diseño de algoritmos eficientes sea la estrategia llamada divide y vencerás, que consiste en la descomposición de un problema en problemas más pequeños, de modo que a partir de la solución de dichos problemas sea posible construir con facilidad una solución al problema completo.

Para ilustrar el método, consideremos el conocido acertijo de las "torres de Hanoi". Consta de tres postes A, B y C. Inicialmente, el poste A tiene cierta cantidad de discos de distintos tamaños, con el más grande en la parte inferior y otros sucesivamente más pequeños encima. El objeto del acertijo es pasar un disco a la vez de un poste a otro, sin colocar nunca un disco grande sobre otro más pequeño, hasta que todos los discos estén en el poste B.

El acertijo puede resolverse con el sencillo algoritmo siguiente. Se imaginan los postes dispuestos en un triángulo. Con un número de movimientos impar, se mueve el disco más pequeño hacia un poste en el sentido de las manecillas del reloj. Con un número de movimientos pares, se hace el único movimiento válido que no implique al disco más pequeño.

El algoritmo anterior es conciso y correcto, pero es difícil entender por qué funciona y de descubrir intuitivamente. En cambio, consideremos el siguiente enfoque que usa la técnica divide y vencerás. El problema de pasar los n discos más pequeños de A a B puede considerarse compuesto de dos problemas de tamaño n-1. Primero se mueven los n-1 discos más pequeños del poste A al C, dejando el n-ésimo disco más

pequeño en el poste A. Se mueve ese disco de A a B. Después se pasan los n-1 discos más pequeños de C a B. El movimiento de los n-1 discos más pequeños se efectúa por medio de la aplicación recursiva del método. Como los n discos implicados en los movimientos son más pequeños que los demás discos, no es necesario preocuparse de los que se encuentran debajo de ellos en los postes A, B o C. Aunque el movimiento real de los discos individuales no es obvio y la simulación a mano es difícil debido al apilamiento de llamadas recursivas, el algoritmo es conceptualmente fácil de entender, de probar que es correcto y, tal vez, de considerarlo como primera opción. Es probable que la facilidad de descubrir los algoritmos de división haga que esta técnica sea tan importante, aunque en muchos casos los algoritmos son también más eficientes que otros más convencionales

7.2 Programación dinámica.

A menudo sucede que no hay manera de dividir un problema en un pequeño número de subproblemas cuya solución pueda combinarse para resolver el problema original. En tales casos, se puede intentar dividir el problema en tantos subproblemas como sea necesario, dividir cada subproblema en subproblemas más pequeños, y así sucesivamente. Si eso es todo lo que se hace, quizá se produzca un algoritmo de tiempo exponencial.

No obstante, con frecuencia, sólo hay un número polinomial de subproblemas, de aquí que se deba resolver algún subproblema muchas veces. Si, en cambio, se conserva la solución a cada subproblema resuelto, y tan sólo se toma la respuesta cuando se requiere, se obtiene un algoritmo de tiempo polinomial.

Desde el punto de vista de la realización, algunas veces es más fácil crear una tabla de las soluciones de todos los subproblemas que se tengan que resolver. Se rellena la tabla sin tener en cuenta si se necesita realmente un subproblema particular en la solución total. La formación de la tabla de subproblemas para alcanzar una solución a un problema dado se denomina programación dinámica, nombre procedente de la teoría de control.

La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común: una tabla a rellenar y un orden en el cual se hacen las entradas.

7.3 Algoritmos voraces.

Consideremos el problema de dar un cambio. Supongamos monedas de 25 centavos (un cuarto) 10 centavos (un décimo), 5 centavos (un vigésimo) y 1 centavo, y que se desea dar un cambio de 63 centavos. Sin pensar, se convierte esta cantidad a dos cuartos, un décimo y tres centavos. No sólo se determinó rápidamente una lista de monedas con el valor correcto , sino que se produjo la lista más corta de monedas con ese valor.

El algoritmo empleado probablemente fue seleccionar la moneda mayor cuyo valor no excedía de 63 centavos (un cuarto), agregarla a la lista y sustraer su valor de 63, quedando 38 centavos. De nuevo, se seleccionó la moneda más grande cuyo valor no fuera mayor de 38 centavos (otro cuarto) y se añadió a la lista, y así sucesivamente.

Este método de dar cambio es un algoritmo voraz. En cualquier etapa individual, un algoritmo voraz selecciona la opción que sea "localmente óptimo" en algún sentido particular. Notemos que el algoritmo voraz para dar cambio produce una solución óptima general debido a las propiedades de las monedas. Si las monedas tuvieran

valores 1 centavo, 5 centavos y 11 centavos y se deseara dar un cambio de 15 centavos, el algoritmo voraz seleccionaría primero la moneda de 11 centavos y después cuatro de 1 centavo, para un total de cinco monedas. Sin embargo, tres monedas de 5 centavos bastan.

Es necesario subrayar el hecho de que no todos los enfoques voraces llegan a dar mejor resultado. Como sucede en la vida real, una estrategia voraz puede dar un buen resultado durante un tiempo, pero el resultado general puede ser pobre.

Para algunos problemas, no existen algoritmos voraces conocidos que produzcan soluciones óptimas; con todo, existen algunos que pueden llegar a producir soluciones "buenas" con una alta probabilidad. A menudo, una solución semióptima con un costo de un pequeño porcentaje sobre la optimalidad es muy satisfactoria. En esos casos un algoritmo voraz con frecuencia brinda la forma más rápida para llegar a una solución "buena". De hecho, si el problema es tal que la única forma de alcanzar una solución óptima es mediante el uso de una técnica de búsqueda exhaustiva, un algoritmo voraz u otro procedimiento heurístico para llegar a una buena solución, pero no necesariamente óptima, puede ser la única opción real.

*7.4 Método de retroceso (***backtracking***)*

Algunas veces surge el problema de encontrar una solución óptima a un subproblema, pero sucede que no existe una teoría que pueda aplicarse para ayudar a encontrar lo óptimo, si no es recurriendo a una búsqueda exhaustiva.

Consideremos un juego como el ajedrez, damas o tres en raya (gato), donde hay dos jugadores. Los jugadores alternan las jugadas, y el estado del juego puede representarse por una posición del tablero. Se hace la suposición de que hay un número finito de posiciones del tablero y que el juego tiene algún tipo de regla para asegurar la terminación. Con cada uno de esos juegos se asocia un árbol llamado árbol de juego. Cada nodo del árbol representa una posición en el tablero, y se asocia la raíz con la posición de partida. Si la posición del tablero x se asocia con el nodo n, los hijos de n corresponderán al conjunto de movimientos posibles desde la posición x del tablero, y cada hijo se asociará con la posición resultante.

Las hojas del árbol corresponden a las posiciones del tablero donde ya no existen movimientos posibles, porque uno de los jugadores ha ganado o porque todos los cuadros están ocupados y se produjo un empate. Se asocia un valor con cada nodo del árbol, y primero se asignan valores a las hojas. Si el juego es tres en raya, una hoja tendrá un valor -1, 0 ó 1, dependiendo de si la posición del tablero corresponde a una derrota, empate o victoria para el jugador 1.

Los valores se propagan hacia arriba en el árbol de acuerdo con la siguiente regla: si un nodo corresponde a una posición del tablero donde hay movimiento del jugador 1, el valor será el máximo de los valores de los hijos de ese nodo. Esto es, se supone que el jugador 1 hará el movimiento más favorable para él, o sea, el que produce el resultado con mayor valor. Si el nodo corresponde al movimiento del jugador 2, entonces el valor será el mínimo de los valores de los hijos. Es decir, se presume que el jugador 2 hará su movimiento más favorable, consiguiendo, de ser posible, una derrota para el jugador 1, o bien un empate como siguiente preferencia.

Notemos que si la raíz tiene valor 1, el jugador 1 tiene una estrategia ganadora; cualquiera que sea su turno, tiene la garantía de seleccionar un movimiento que le dé una posición del tablero con valor 1. Cuando el turno corresponde al jugador 2, no tiene

más opción que seleccionar un movimiento que deje una posición del tablero con valor 1, lo cual significa una derrota para él. El hecho de que se suponga que el juego termina, garantiza que el primer jugador podrá ganar. Si la raíz tiene valor 0, como sucede en el juego tres en raya, ningún jugador tiene una estrategia ganadora, pero por lo menos puede garantizarse un empate si juega lo mejor posible. Si la raíz tiene valor -1, entonces el jugador 2 tiene una estrategia ganadora.

7.4.1 Funciones de utilidad.

La idea de un árbol de juego, donde los nodos tienen valores -1, 0 y 1, puede generalizarse a árboles donde a las hojas se les da un número cualquiera (llamado la utilidad) como valor, y se aplican las mismas reglas de evaluación de nodos interiores: toma el máximo de los hijos en los niveles donde corresponda un movimiento del jugador 1, y el mínimo de los hijos en los niveles donde mueve el jugador 2.

Como ejemplo de donde son provechosas las utilidades generales, consideremos un juego complejo como el ajedrez, donde el árbol de juego, aunque finito, es tan inmenso que la evaluación de abajo arriba no es posible. Los programas de ajedrez operan construyendo el árbol de juego con ese tablero como raíz para cada posición del tablero desde la cual se debe mover extendiéndose hacia abajo por varios niveles; el número exacto de niveles depende de la velocidad con que el computador puede trabajar. Puesto que la mayor parte de las hojas del árbol son ambiguas, no indican triunfos, derrotas, ni empates, cada programa usa una función de las posiciones del tablero que intenta evaluar la probabilidad de que el computador gane desde esa posición. Por ejemplo, la diferencia entre las piezas afecta en gran medida tal estimación, al igual que algunos factores, como el poder defensivo alrededor de los reyes. Al usar esta función de utilidad, el computador puede estimar la probabilidad de un triunfo después de hacer cada uno de sus siguientes movimientos posibles en el supuesto de que se hará la mejor jugada siguiente de cada lado, y se escogerá el movimiento de mayor utilidad.

7.4.2 Realización de la búsqueda con retroceso.

Supongamos que se proporcionan las reglas de un juego, esto es, sus movimientos válidos y reglas de terminación. Se desea construir su árbol de juego y evaluar la raíz, en la manera más obvia, y después visitar los nodos en orden final. El recorrido en orden final asegura que se visita un nodo interior n después de sus hijos, y entonces es posible evaluar n, tomando el mínimo o el máximo de los valores de sus hijos, lo que sea más apropiado.

El espacio de almacenamiento del árbol puede ser demasiado grande, pero con cuidado nunca se necesitará almacenar más de un camino, desde la raíz hasta un nodo, en cualquier momento. Un programa que implemente esto supone lo siguiente:

1. Las utilidades son números reales en un intervalo limitado, por ejemplo –1 a +1.
2. La constante ∞ es mayor que cualquier utilidad positiva y su negativo es menor que cualquier utilidad negativa.
3. El tipo tipo_modo se declara como: type tipo_modo = (MIN, MAX)
4. Existe un tipo tipo_tablero declarado de alguna forma adecuada para la representación de posiciones en el tablero.

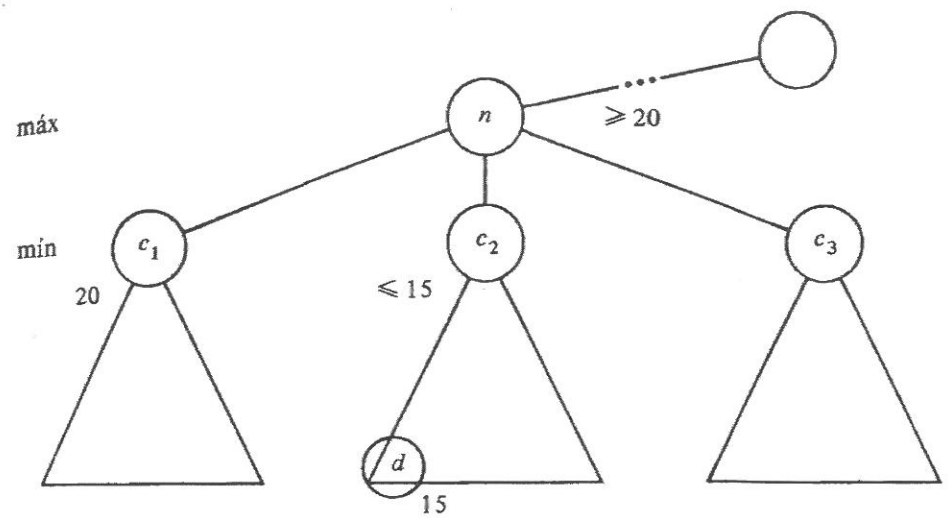
5. Existe una función UTILIDAD que calcula la utilidad de cualquier tablero que sea una hoja (esto es, situación de triunfo, derrota o empate).

```
funcion busca (B: tipo_tablero; modo: tipo_modo ) : real;
var C: tipo_tablero;
    valor: real;
begin
   if B es una hoja then
      return (utilidad(B))
   else begin
      if modo = MAX then
         valor:= -∞
      else
         valor:= ∞;
      for cada hijo C del tablero B do
         if modo = MAX then
            valor:= máx(valor, busca(C, MIN))
         else
            valor:= mín(valor, busca(C, MAX));
      return (valor)
   end
end;
```

Otra implementación a considerar es por medio de un programa no recursivo que contenga una pila de tableros correspondientes a la secuencia de llamadas activas de busca.

7.4.3 Poda alfa-beta.

Hay una observación simple que permite dejar de considerar buena parte de un árbol de juego típico. El ciclo for puede saltar ciertos hijos, con frecuencia muchos. Supongamos que se tiene un nodo *n*, como en la siguiente figura:



Ya se ha determinado que c_1, el primero de los hijos de *n*, tiene valor 20. Como *n* es un nodo máx, se sabe que su valor es por lo menos 20. Ahora, supongamos que al continuar con la búsqueda se observa que *d*, un hijo de c_2, tiene valor 15. Como c_2, otro hijo de *n*, es un nodo mín, se sabe que el valor de c_2 no puede exceder de 15. Así, cualquiera que sea el valor c_2 no afecta al valor de *n* ni de cualquier padre de *n*.

Así, es posible ignorar la consideración de los hijos de c_2 que aún no se han examinado. Las reglas generales para pasar por alto o "podar" nodos implican la noción de valores tentativos y finales para los nodos. El valor final es el que se ha denominado

hasta ahora simplemente "valor". Un valor tentativo es una cota superior al valor del nodo mín, o una cota inferior al valor del nodo máx. Las reglas para el cálculo de los valores finales y tentativos son las siguientes:

1. Si todos los hijos de un nodo n se consideraron o se podaron, conviértase el valor tentativo de *n* en final.
2. Si un nodo máx *n* tiene valor tentativo v_1 y un hijo con valor final v_2, entonces el valor tentativo de *n* se hace máx(v_1, v_2). Si *n* es un nodo mín, se asigna su valor tentativo a mín(v_1, v_2).
3. Si *p* es un nodo mín con padre *q* (un nodo máx), y *p* y *q* tienen valores tentativos v_1 y v_2, respectivamente, con $v_1 \leq v_2$, entonces se pueden podar todos los hijos no considerados de *p*. También se pueden podar los hijos no considerados de *p* si *p* es un nodo máx (y por tanto, *q* es un nodo mín) y $v_2 \leq v_1$.

8 Conclusiones.

Un algoritmo es una secuencia de operaciones bien definidas que resuelven un problema, donde cada una de dichas secuencias de operaciones requiere una cantidad finita de memoria y se realiza en un tiempo finito.

Existen diversas técnicas para representar algoritmos: pseudocódigo, organigramas, diagramas de Nassi-Schneiderman, y notaciones tabulares de diseño de algoritmos como por ejemplo las tablas de decisión. En cualquier caso, cualquiera de estas técnicas, si se usa correctamente, puede ser una ayuda muy valiosa en el proceso de diseño.

Entre las distintas acciones que pueden utilizarse en un algoritmo, y las estructuras que se pueden utilizar para controlar el orden de éstas tenemos: asignaciones, entradas/salidas, decisiones, ciclos, y procedimientos o subrutinas.

El diseño descendente es una técnica natural de desarrollo de algoritmos. Consiste en comenzar trabajando a nivel abstracto, para ir dividiendo el problema en sus partes naturales, de manera que el problema se descompone en otros más simples. A éstos se les aplica el mismo procedimiento hasta llegar a problemas suficientemente pequeños como para ser resueltos directamente.

En cuanto a las técnicas más habituales para el desarrollo de algoritmos, tenemos: divide y vencerás, programación dinámica, algoritmos voraces, y el método de retroceso (*backtracking*) que utiliza recursividad.