**UNIVERSIDAD TECNOLÓGICA NACIONAL**

**FACULTAD REGIONAL RESISTENCIA**



**INGENIERÍA EN SISTEMAS DE INFORMACIÓN**

**GESTIÓN DE DATOS**

APUNTES TEÓRICOS

**Unidad 1: Introducción**

**Profesor Teoría:** Ing. Carolina Orcola

**Jefe de T. P.:** Ing. Luis Eiman

**Auxiliar de T.P.:** Juan Carlos Fernandez



Índice

Unidad 1: Introducción ....................................................................................................................3

1- Complejidad Algorítmica .........................................................................................................3

1.1 Algoritmia ..........................................................................................................................3

1.1.1 Fases en el desarrollo de un algoritmo: ......................................................................4

1.1.2 Verificación y Análisis de Algoritmos ..........................................................................4

1.2 Análisis de la Eficiencia de Algoritmos ..............................................................................4

1.2.1 Tiempo de ejecución y uso de memoria .....................................................................4

1.2.2 Análisis de un Algoritmo .............................................................................................5

1.2.3 Comportamiento asintótico .........................................................................................8

1.2.4 Funciones de complejidad en tiempo más usuales.....................................................9

1.2.5 Eficiencia versus claridad ........................................................................................ 10

1.3 Elección de un Algoritmo................................................................................................. 10

1.4 Problemas tratables e intratables .................................................................................... 11

2- Subalgoritmos o Subprogramas............................................................................................ 12

2.1 Funciones ....................................................................................................................... 12

2.2 Procedimientos ............................................................................................................... 13

2.3 Ámbito: Variables Locales y Globales ............................................................................. 13

2.4 Comunicación con subprogramas: Paso de parámetros ................................................. 14

2.4.1 Paso por valor .......................................................................................................... 14

2.4.2 Paso por parámetro.................................................................................................. 14

2.4.3 Síntesis de la transmisión de parámetros ................................................................. 15

3- Recursividad ......................................................................................................................... 16

3.1 ¿Cuándo no usar la recursión? ....................................................................................... 17

3.2 Algoritmo Divide y Vencerás ........................................................................................... 18

3.3 Algoritmo de vuelta atrás (Backtracking) ......................................................................... 18

3.4 Problema de la selección óptima..................................................................................... 19

3.5 Problema de los Matrimonios Estables ........................................................................... 20

4- Técnicas de diseño de algoritmos......................................................................................... 21

4.1 Algoritmos de fuerza bruta .............................................................................................. 21

4.2 Algoritmos ávidos o voraces ........................................................................................... 21

4.3 Divide y vencerás y Algoritmos con Retroceso (backtracking)......................................... 22

4.4 Programación dinámica................................................................................................... 22

4.5 Algoritmos probabilistas .................................................................................................. 22

5- Bibliografía............................................................................................................................ 23



**Unidad 1: Introducción**

Nuestra materia se centra en el “buen” diseño y elección de estructuras de datos y algoritmos. Dicho de manera simple, una ***estructura de datos*** es una forma sistemática de organizar y dar acceso a los datos, y un ***algoritmo*** es un procedimiento detallado para hacer alguna tarea en una cantidad finita de tiempo. Pero para calificar como “buenas” algunas estructuras de datos y algoritmos debemos contar con alguna forma precisa de analizarlas.

**1- Complejidad Algorítmica**

***1.1 Algoritmia***

Dimos ya una definición de algoritmo. Otras pueden ser:

Ø Un *algoritmo* es una secuencia finita de instrucciones que especifican cómo se resuelve un problema.

Ø Un *algoritmo* es cualquier procedimiento computacional bien definido, junto con un conjunto especificado de datos, que produce un valor o conjunto de valores como salida.

Esta última definición resalta que un algoritmo debe producir una salida válida para todas las entradas permitidas.

Pero, ¿qué es la algoritmia?

Muchas veces contamos con muchos algoritmos diferentes que resuelven un mismo problema, entonces surge la pregunta de ¿cuál usar?. Para responder a esta pregunta está la algoritmia.

La ***algoritmia*** es la ciencia que estudia técnicas para diseñar algoritmos eficientes y evaluar la eficiencia de un algoritmo que ya existe.

Un algoritmo es mucho mejor cuantos menos recursos consuma, eso es evidente, pero se deben tener en cuanta otros factores antes de establecer la conveniencia o no de este. Por ejemplo, la facilidad para programarlo, la facilidad de entenderlo, la robustez, etc., y por supuesto, el criterio de eficiencia buscado: relación entre recursos consumidos y productos conseguidos.

De forma general podemos decir que los recursos que consume un algoritmo dependen de factores internos y externos.

Ø Factores internos

Tamaño de los datos de entrada

Naturaleza de los datos de entrada (peor caso, caso promedio o mejor caso)

Ø Factores externos

El ordenador donde lo ejecutamos: procesador, memoria, etc.. El lenguaje de programación y el compilador usado.

La implementación que haga el programador del algoritmo, en particular de las estructuras de datos usadas.

De estos dos factores los externos no aportan información sobre el algoritmo, porque no pueden ser controlados por el usuario. Mientras que los factores internos, el tamaño y contenido de los datos de entrada, si pueden ser controlados por el usuario, por lo que aportan la información necesaria para poder realizar el análisis del algoritmo.

La ***algoritmia*** es la ciencia que permite evaluar el efecto de los diferentes factores internos sobre los algoritmos, de tal modo que sea posible seleccionar el que más se ajuste a nuestras circunstancias, además de indicar la forma de diseñar un nuevo algoritmo para una tarea concreta.



***1.1.1 Fases en el desarrollo de un algoritmo:***

Cualquier consideración sobre el desarrollo de un algoritmo con cierta complejidad conceptual debe comenzar aislando cada una de sus fases componentes. Se pueden identificar las siguientes etapas de una manera más o menos general.

1. *Análisis del problema:* Se refiere a la etapa en la cual el programador toma conocimiento del problema antes de proceder a desarrollar una solución. Un análisis inadecuado puede conducir a una mala interpretación del enunciado del problema. Los errores en esta etapa son, con frecuencia, difíciles de detectar y consumen mucho tiempo al arrastrarse hacia fases posteriores.

2. *Desarrollo de la solución:* Una vez definido el problema y teniendo cierta idea de cómo resolverlo, se puede utilizar alguna de las técnicas conocidas de diseño de algoritmos. Muchas veces, debido a la complejidad interna de los problemas a resolver, se puede ir describiendo la solución como una secuencia de pasos bastante generales (esto puede hacerse en lenguaje natural), que cada vez se van detallando o refinando más hasta obtener una solución. En esta etapa también se empiezan a tomar decisiones sobre las estructuras de datos que se utilizarán para representar los datos del problema.

3. *Codificación de la solución:* Considerando que la solución algorítmica ha sido bien definida, este proceso resulta casi completamente mecánico. Utilizando las reglas sintácticas y semánticas de un lenguaje de programación, el algoritmo se escribe teniendo en cuenta también ciertos criterios de estilo o estructura.

4. *Verificación y análisis de la solución:* Aquí debemos tener en cuenta tres aspectos claves del algoritmo: su corrección o eficiencia, su claridad o legibilidad y su eficacia o uso óptimo de los recursos.

***1.1.2 Verificación y Análisis de Algoritmos***

Es habitual considerar como objetivos de la programación los de CORRECCIÓN, CLARIDAD y EFICIENCIA. De acuerdo con ello, para evaluar un algoritmo o programa podemos tener en cuenta estos tres aspectos:

ü CORRECCIÓN o EFICACIA: Si el algoritmo cumple con las especificaciones dadas y funciona para todas las entradas permitidas.

ü CLARIDAD: Si el algoritmo es fácil de codificar y depurar o entender, incluso por personas distintas de su diseñador.

ü EFICIENCIA: Si el algoritmo resuelve el problema usando una cantidad razonable de recursos.

Los objetivos anteriores resultan a veces contradictorios. Por supuesto, el objetivo de corrección es fundamental, pero los otros dos han de satisfacerse a veces estableciendo un compromiso entre ellos. En muchos casos podemos tener un algoritmo complejo pero muy eficiente, frente a un algoritmo sencillo pero que use muchos recursos.

El análisis de la corrección de un algoritmo se denomina *verificación*. La claridad de la implementación es difícil de medir de manera objetiva, de manera que no hay técnicas formales para ello. El análisis de eficiencia conduce a las medidas de complejidad algorítmica.

***1.2 Análisis de la Eficiencia de Algoritmos***

Analizar la *eficiencia* de un algoritmo supone determinar la cantidad de recursos informáticos consumidos por el algoritmo. Los recursos que habitualmente se contabilizan incluyen la cantidad de memoria y la cantidad de tiempo de cómputo requeridos por el algoritmo.

***1.2.1 Tiempo de ejecución y uso de memoria***

Nuestro análisis de los algoritmos y estructuras de datos se centrará principalmente en consideraciones de eficiencia. Este análisis se realiza considerando la cantidad de recursos que un algoritmo consume, como una función del tamaño de la entrada del algoritmo.



La *memoria* consumida por un algoritmo es la suma de dos componentes: una *parte fija* que es

independiente de sus datos y resultados, y una *parte variable* que depende del tamaño del problema considerado.

La *memoria fija* consiste en el espacio reservado para el código del algoritmo, variables simples, constantes, estructuras de datos de tamaño fijo, etc.

La *memoria variable o dinámica* es el espacio necesitado por las variables cuyo tamaño dependa de la instancia del problema que está siendo resuelta, del espacio requerido (si es preciso) para las diferentes activaciones de los subprogramas recursivos, etc..

El *tiempo de ejecución* para un programa depende de factores como los datos de entrada, la calidad del código generado por el compilador, la naturaleza y velocidad de las instrucciones de máquina empleadas en la ejecución y la complejidad en el tiempo del algoritmo utilizado en el programa.

En muchos casos, el espacio y el tiempo requeridos por un algoritmo están inversamente relacionados. Esto es, podemos ser capaces de reducir los requisitos de espacio en memoria con un incremento del tiempo de ejecución, o a la inversa, reducir los requisitos de tiempo incrementando el espacio en memoria.

***1.2.2 Análisis de un Algoritmo***

Si se ha implementado el algoritmo se puede estudiar su tiempo de ejecución, corriéndolo con diversas entradas de prueba y anotando el tiempo real que se uso en cada ejecución. Esas medidas se pueden tomar en forma exacta, usando llamadas de sistema que se incorporan al lenguaje o al sistema operativo para el que se escribió el algoritmo (por ejemplo usando el método System.currentTimeMillis()). En general, lo que interesa es determinar la dependencia entre el tiempo de ejecución y el tamaño de la entrada.

Para determinarlo, se pueden hacer algunos

experimentos con muchas entradas de prueba sistintas, de varios tamaños; y luego visualizar los resultados de esos experimentos, graficando el funcionamiento de cada corrida del algoritmo como un punto con abscisa x igual al tamaño de la entrada, n, y cuya ordenada sea el tiempo de ejecución, t.

**Tiempo (ms)**

Para que esto tenga sentido, se requiere que para este análisis se escojan entradas de muestra buenas, y que se prueben las suficientes como para establecer afirmaciones estadísticas acerca del algoritmo.

El tiempo de ejecución de un programa depende, en la mayoría de los casos, de los datos particulares con los que opera. Esto quiere decir que la eficiencia de un programa debe establecerse no como una magnitud fija

**9000**

**8000**

**7000**

**6000**

**5000**

**4000**

**3000**

**2000**

**1000**

**0**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

**0 50 100**

**Tamaño de la Entrada**

para cada programa, sino como una función que nos dé el tiempo de ejecución para cada tamaño o cantidad de los datos que deba procesar.

Esta idea nos lleva, por su parte, a la necesidad de establecer previamente una medida del tamaño de los datos o *tamaño del problema*, para, en función de ella, establecer la medida de la eficiencia del programa que los procesa.

El tamaño del problema se puede expresar bien por la cantidad de datos a tratar, o bien por los valores particulares de los datos. Por ejemplo, para un programa que obtenga el valor medio o la suma de una serie de datos, la cantidad de números a sumar o promediar es una buena medida del tamaño del problema. En cambio, para un programa que calcule una potencia de un número, el tamaño significativo puede ser el valor del exponente más que el del número original.



La complejidad algorítmica en tiempo se mide como la función que da el tiempo de ejecución

según el tamaño del problema. Denotamos a esa función como *T(n).*

*T(n)= cf(n)*

Cuando decimos que la complejidad de un algoritmo es *T(n) = cf(n)* no estamos haciendo referencia a una determinada unidad de tiempo, sino a que el tiempo de ejecución del algoritmo es proporcional a cierta función *f(n)*, donde la constante de proporcionalidad *c* depende del

computador.

En general se tienen tres criterios para calcular *T(n):*

ü Considerar *T(n)* como el tiempo para el *peor caso* de los posibles.

ü Considerar *T(n)* como el tiempo para el *mejor caso* de los posibles.

ü Hacer *T(n)* igual al *tiempo medio* de todos los casos posibles.

A primera vista, es mejor el caso promedio pero hay que tener en cuenta que no todos los casos posibles son equiparables. Si no se tiene en cuenta los efectos probabilísticos en el cálculo de *T(n)*, esta medida puede resultar errónea. La primera medida también puede no resultar totalmente correcta pero es más segura y fácil de evaluar, por ello nos centrarnos en el tiempo de ejecución en el peor

**120**

**100**

**Tiempo de ejecución**

**80**

**60**

**40**

**20**

**0**

mejor caso

caso promedio peor caso

**1000 2000 3000 4000**

**Tamaño de la entrada**

caso. Además, es crucial en aplicaciones como juegos, finanzas y robótica.

Si bien los estudios experimentales de los tiempos de ejecución tienen utilidad, tienen también tres limitaciones importantes:

Ø Es necesario implementar y ejecutar un algoritmo, para estudiar en forma experimental su tiempo de ejecución.

Ø Es difícil comparar la eficiencia de dos algoritmos, a menos que se hayan hecho experimentos para conocer sus tiempos de ejecución en los mismos ambientes de hardware, de software y con las mismas muestras de entradas.

Ø Solo se pueden hacer experimentos con un conjunto limitado de entradas de prueba, y los experimentos pueden no ser indicativos del tiempo de ejecución con otras entradas que no se hayan incluido en el experimento.

Por esto adoptamos una metodología general para analizar el tiempo de ejecución de algoritmos que:

Ø Considera todas las entradas posibles.

Ø Permite evaluar la eficiencia relativa de dos algoritmos cualesquiera en una forma que es independiente del ambiente del hardware y del software.

Ø Puede ejecutarse estudiando una descripción de alto nivel del algoritmo, sin implementarlo o experimentar con él en realidad.

Por tanto, para la evaluación de la complejidad en tiempo se hace referencia a una máquina virtual hipotética con tiempo de ejecución estándar, donde consideramos que cada operación elemental del lenguaje de programación: suma, resta, división, multiplicación, escritura, asignación, decisión según condición, llamada a un método (función o procedimiento), regreso de un método, etc., dura una unidad de tiempo.

Con esta simplificación, el análisis de la eficiencia de un programa se centra en establecer cuántas instrucciones se ejecutan en total, dependiendo del tamaño o cantidad de los datos a procesar.



Usando el criterio del *peor caso* en análisis de algoritmos, debemos tener en cuenta las siguientes

reglas:

Regla 1) La complejidad de un esquema secuencial es la suma de las complejidades de sus acciones componentes.

Regla 2) La complejidad de un esquema de selección equivale a la de la alternativa más compleja, es decir, de ejecución más larga, más la complejidad de la evaluación de la condición de selección.

Regla 3) La complejidad de un esquema de iteración se obtiene sumando la serie correspondiente al número de instrucciones en las repeticiones sucesivas.

Veamos cómo es este análisis en algunos casos concretos. Tomemos como ejemplo el siguiente fragmento de programa que obtiene el máximo de dos números:

*máximo := a; Si a < b*

*entonces máximo: = b;*

El esquema global es una secuencia de dos acciones: una asignación, seguida de un esquema condicional (Si). Anotaremos el programa con el número correspondiente a cada sentencia, para lo cual contaremos el número de operadores (+,-,\*,<,:=,etc.) y decisiones (Si, Mientras, etc.). Aplicando las reglas para los esquemas tendremos:

*máximo := a;* 1

*Si a < b* 2

1 + 1 *entonces máximo := b;* 1

3 (regla 2)

*T(n)* = 4

Total = 4 (regla 1)

La complejidad en este caso es fija y no depende de una medida de tamaño del problema.

A continuación analizaremos un bucle que obtiene en *f* el factorial de un número *k.* Anotaremos el programa con el número de instrucciones de cada sentencia.

*k:= 1;* 1

*f:= 1;* 1

*Mientras k<n hacer* 2

*k:= k + 1;* 2 6(n-1) (Regla3)

|  |  |  |
| --- | --- | --- |
|  | |  |
|  |  | |

*f:= f \* 1;* 2

*fin-Mientras;*

T(n) = 6n - 4

Total = 6n – 4 (Regla1)

Para calcular el número de instrucciones del bucle se ha multiplicado el número de instrucciones en cada repetición por el número de repeticiones.

La complejidad aparece expresada en función del valor de *n,* que en este caso resulta una medida natural del tamaño del problema

Considérense los siguientes fragmentos de algoritmos.

1) *x : = x + 1;*

2) *Para i de 1 a n hacer x : = x + 1;*

*fin-Para;*

T(n)= 2

T(n)= 2n



3) *Para i de 1 a n hacer*

*Para j de 1 a n hacer x : = x + 1 ;*

*fin-Para;*

*fin-Para;*

T(n)= 2n2

En el punto número (1), la sentencia de asignación se ejecuta sólo una vez; en la (2), la misma sentencia de ejecuta *n* veces; en la (3), se ejecuta *n*² veces (por existir dos bucles anidados en *n* repeticiones cada uno). Por ejemplo, para n = 10 corresponderían a 1, 10 y 100 repeticiones, respectivamente. Estos valores permiten diferenciar los distintos órdenes de complejidad de cada uno de los algoritmos.

Si hay llamadas a métodos (funciones o procedimientos), éstas deben ser analizadas primero y sumadas luego. Por ejemplo:

cont:= 0;

*x : = 1;*

*Mientras x < n hacer x:= x +1;*

*cont:= cont + Proceso(x, n); Fin\_mientras*

1

1

2

2

2 + 1 + T(Proceso(x, n)) n veces suponemos T()= 2n

T(n)= 2 + 7n + 2n2

Si hay procedimientos recursivos, hay varias opciones. Si la recursión es un ciclo *for* ligeramente disfrazado, el análisis suele ser trivial. Por ejemplo, la siguiente función es un ciclo sencillo:

Función **factorial** (n: entero): entero; Si n=0 o n=1

entonces factorial:= 1;

sino factorial:= n \* **factorial** (n-1); Fin\_función

Este ejemplo es realmente un uso ineficiente de la recursión. Cuando la recursión se usa

adecuadamente, es difícil convertirla en una estructura iterativa sencilla. En este caso, el análisis implicará una relación de recurrencia que hay que resolver, y el tiempo de ejecución es logarítmico.

***1.2.3 Comportamiento asintótico***

En los análisis de eficiencia o complejidad se considera muy importante la manera en que la función de complejidad va aumentando con el tamaño del problema. *Lo que interesa es la forma de crecimiento del tiempo de ejecución, y no tanto el tiempo particular empleado*.

Como ejemplo podemos comparar dos programas, uno que tarda un tiempo 100n en resolver un problema de tamaño n, y otro que tarda un tiempo n2. La comparación puede hacerse escribiendo una tabla con los tiempos de cada uno para diferentes tamaños del problema.

|  |  |  |
| --- | --- | --- |
| Tamaño | 100n | n2 |
| 1 | 100 | 1 |
| 2 | 200 | 4 |
| 3 | 300 | 9 |
| 10 | 1.000 | 100 |
| 100 | 10.000 | 10.000 |
| 1000 | 100.000 | 1.000.000 |

Al principio de la tabla el primer programa parece menos eficiente que el segundo, ya que tarda mucho más tiempo, pero a medida que aumenta el tamaño del problema ambos programas llegan a tardar lo mismo (para tamaño 100) y a partir de ahí el segundo programa demuestra ser mucho menos eficiente que el primero.

La menor eficiencia del segundo programa para tamaños grandes del problema no cambia por el hecho de que se modifique el coeficiente multiplicador; es decir, si el primer programa tardase 10



veces más (1000n en lugar de 100n), acabaría igualmente por resultar mejor que el segundo a

partir de un cierto tamaño del problema.

Lo que importa es la forma de la función, que en el primer caso es *lineal* y en el segundo es *cuadrática*. La forma en que crece la función para tamaños grandes se dice que es de **crecimiento asintótico**, y se representa mediante la notación o grande:

*O(g(n))*

Siendo *n* el tamaño del problema, *g* la forma o función de crecimiento asintótico, y O el **orden de crecimiento**.

Sea una función *f(n)*= tiempo de ejecución del algoritmo, decimos que dicha función es *O(g(n))* si se cumple que existe una constante *c* > 0 y un número n0 (que no depende de n) tales que:

f(n) / g(n) c para todo n0 n

Ejemplos:

Si T(n)= 4n3 + 2n2 tenemos que T(n) es O(n) ya que tomando *c* = 5 y n0=2 obtenemos: (4n3 + 2n2) / n3 5 para todo n0 n

entonces O (n3)

***1.2.4 Funciones de complejidad en tiempo más usuales***

Las funciones de complejidad algorítmica más usuales, ordenadas de mayor a menor eficiencia son:

|  |  |  |
| --- | --- | --- |
| **O (1)** | *complejidad constante* | Es la situación más deseada pero difícil de encontrar en la mayoría de los problemas. |
| **O(*log n*)** | *complejidad logarítmica* | Esta complejidad suele aparecer en determinados algoritmos recursivos, en aquellos en los que el uso de la recursión es aconsejable. Es una complejidad óptima. |
| ***O(n)*** | *complejidad lineal* | Es en general, una complejidad buena y bastante usual. Suele aparecer en la evaluación de un bucle simple cuando la complejidad de las operaciones interiores es constante o en algoritmos con recursión estructural (simples de pasar a iterativos). |
| **O(n log n)** | *complejidad logarítmica* | También aparecen en algoritmos con recursión no estructural (por ejemplo, ordenación por el método rápido) y se considera una complejidad buena. |
| **O(*n²)*** | *complejidad cuadrática* | Aparece en bucles dobles o similares. |
| **O(*n³)*** | *complejidad cúbica* | Aparece en bucles triples. Para un valor grande de *n*  empieza a crecer en exceso. |
| **O(*nk)*** | *complejidad polinómica*  *(k > 3)* | Si *k* crece, la complejidad del programa es bastante mala. |
| **O(*2n)*** | *complejidad exponencial* | Debe evitarse en la medida de lo posible. Puede aparecer en un subprograma recursivo que contenga dos o tres llamadas internas. En programas donde aparece esta complejidad suele hablarse de *explosión combinatoria.* |

Téngase en cuenta que aunque un algoritmo posea una eficiencia que en términos absolutos pueda calificarse como “buena”, esto no significa que lo sea en términos relativos, pues puede ocurrir que la naturaleza del problema permita una solución más eficiente (esto es, de menor complejidad).



La tabla siguiente muestra el crecimiento de las diferentes funciones según el parámetro *n*.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | log n | n log n | n2 | n3 | 2n |
| 1 | 0 | 0 | 1 | 1 | 2 |
| 5 | 0.7 | 3 | 25 | 125 | 32 |
| 10 | 1.0 | 10 | 100 | 1000 | 1024 |
| 20 | 1.3 | 26 | 400 | 8000 | 1048576 |
| 50 | 1.7 | 85 | 2500 | 25000 | 1125900000x106 |
| 100 | 2.0 | 200 | 10000 | 1000000 | 1267651000x1021 |
| 200 | 2.3 | 460 | 40000 | 8000000 | - |
| 500 | 2.7 | 1349 | 250000 | 125000000 | - |
| 1000 | 3.0 | 3000 | 1000000 | 1000000000 | - |

***1.2.5 Eficiencia versus claridad***

Las grandes velocidades de los microprocesadores actuales, junto con el aumento considerable de las memorias centrales, y los bajos precios de los mismos, hacen que los recursos típicos tiempo y almacenamiento no sean hoy en día parámetros fundamentales para la medida de la eficiencia de un programa.

Por otra parte es preciso tener en cuenta que a veces los cambios para mejorar la eficiencia de un programa pueden traducirse en menor legibilidad o menor claridad; lo cual hace que una modificación o agregado luego sea más dificultoso. En programas grandes la legibilidad suele ser más importante que el ahorro en tiempo y en almacenamiento en memoria. Como norma general, cuando la elección en un programa se debe hacer entre claridad y eficiencia, generalmente se elegirá la claridad del programa.

***1.3 Elección de un Algoritmo***

ü Para resolver un problema debemos encontrar un algoritmo que tenga una complejidad lo más pequeña posible dentro de la tabla anterior. Si dudamos entre dos algoritmos de la misma complejidad podemos calcular la complejidad en media o hacer una prueba en el ordenador.

ü Si un programa va a ser usado un número pequeño de veces se puede decir (salvo que utilice una cantidad excesiva de recursos) que el costo de escribir el programa y de depurarlo domina. Por tanto se optará por el algoritmo más sencillo de realizar correctamente.

ü Si el programa va a ser utilizado con una cantidad de datos relativamente pequeña, debemos darnos cuenta que la notación O() no es muy significativa. En este caso, si es posible, se puede acudir al tiempo medio o a una comprobación empírica del comportamiento de los algoritmos.

ü Un algoritmo eficiente pero excesivamente complicado puede no ser muy útil, sobre todo si el algoritmo va a ser utilizado por otra persona que no sea el diseñador.

ü Los accesos a memorias secundarias suelen ser muy lentos. Deben, por tanto, reducirse a las imprescindibles.

ü En algoritmos de análisis numéricos en problemas matemáticos puede se necesario usar algoritmos más complejos para satisfacer la exigencia de precisión requerida.



***1.4 Problemas tratables e intratables***

El hecho de que un problema sea resoluble en teoría no quiere decir que se pueda siempre resolver en la práctica. Desde un punto de vista pragmático, sólo se puede considerar resuelto un problema si se encuentra un algoritmo suficientemente eficiente para obtener la solución.

Problema Resoluble:

En teoría --------> Existe algoritmo

En la práctica ---> Existe algoritmo y es eficiente

En la mayoría de los casos lo que puede ocurrir es que un determinado problema sólo sea resoluble en la práctica para tamaños pequeños de los datos. Esto ocurre con aquellos problemas para los que sólo se conocen algoritmos de complejidad elevada.

Los problemas resolubles en teoría pero no en la práctica por la ineficiencia de los algoritmos conocidos, se denomina problemas intratables. Los textos de complejidad algorítmica establecen normalmente como límite de eficiencia de los problemas tratables el correspondiente a un crecimiento asintótico de tipo potencia de exponente constante O(nk). Este crecimiento se da en los algoritmos cuya función de coste es de tipo polinómico. Los problemas que pueden resolverse con algoritmos de complejidad polinómica se consideran tratables.

Los problemas para los que sólo existen algoritmos menos eficientes que los de complejidad polinómica (por ejemplo exponencial O(kn)) se consideran problemas intratables.



**2- Subalgoritmos o Subprogramas**

Refrescando conceptos ya vistos en años anteriores, recordemos que un subprograma puede realizar las mismas acciones que un programa: 1) aceptar datos; 2) realizar cálculos; y 3) devolver resultados. Sin embargo un subprograma se utiliza por un programa para un propósito específico. El subprograma recibe datos desde el programa y le devuelve resultados.

Se dice que el programa principal *llama* o *invoca* al subprograma. El subprograma ejecuta una tarea específica y *devuelve* el control al programa. Esto puede suceder en diferentes lugares del programa, y a su vez un subprograma puede llamar a sus propios subprogramas.

Programa

Llamada 1

Retorno 1

Llamada 2

Retorno 2

Subprograma 1

Subprograma 1.1

Existen, como recordarán, dos tipos de subprogramas: *funciones* y *procedimientos*.

***2.1 Funciones***

Matemáticamente una función es una operación que toma uno o más valores llamados *argumentos* y produce un valor denominado *resultado* –valor de la función para los argumentos dados-. Todos los lenguajes de programación tienen *funciones incorporadas* o *intrínsecas* y funciones definidas por el mismo usuario, o *funciones externas*.

Cuando las funciones estándares o internas no permiten realizar el tipo de cálculo deseado es necesario recurrir a las funciones externas, que pueden ser definidas por el usuario mediante una *declaración de función*.

A una función no se la llama explícitamente, sino que se la invoca o referencia mediante un nombre y una lista de parámetros actuales.

El algoritmo o programa llama o invoca a la función con su nombre en una expresión seguida de una lista de argumentos que deben coincidir en cantidad, tipo y orden con los de la función que fue definida. La función devuelve un único valor.

Una función puede ser llamada de la siguiente forma:

**Nombre\_función** (*lista de parámetros actuales*)

Los argumentos de la declaración de la función se denominan *parámetros formales*, *ficticios* o *mudos*; son nombres de variables de otras funciones o procedimientos y solo se utilizan dentro del cuerpo de la función. Los argumentos usados en la llamada a la función se denominan *parámetros actuales*, que a su vez pueden ser variables, constantes, expresiones, valores de funciones o nombres de otras funciones.

Cada vez que se llama a la función desde el programa principal se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función y se presupone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

Una llamada a la función implica los siguientes pasos:

1) A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual.

2) Se ejecuta el cuerpo de acción de la función.

3) Se devuelve el valor de la función al nombre de la función y se retorna al punto de llamada.



***2.2 Procedimientos***

Aunque las funciones son herramientas de programación muy útiles para la resolución de problemas, su alcance está muy limitado. Con frecuencia se requieren subprogramas que calculen varios resultados en vez de uno solo, o que realicen la ordenación de una serie de números, etc. En estas situaciones una *función* no es apropiada y se necesita disponer del otro tipo de subprogramas: los *procedimientos*.

Un procedimiento es un subprograma que ejecuta un proceso específico. Ningún valor está asociado con el nombre del procedimiento; por consiguiente no puede ocurrir en una expresión. Un procedimiento se llama escribiendo su nombre seguido de los parámetros actuales

Los parámetros formales de la declaración del procedimiento tienen el mismo significado que en las funciones; los parámetros variables –en aquellos lenguajes que lo soportan, por ej. Pascal- están precedidos cada uno de ellos por la palabra **var** para designar que ellos obtendrán resultados del procedimiento en lugar de los valores actuales asociados a ellos.

La lista de parámetros, bien formales en el procedimiento o actuales en la llamada al mismo, se conocen como lista de parámetros, y estas pueden ser de Entrada, de Salida o de Entrada/Salida.

***2.3 Ámbito: Variables Locales y Globales***

Las variables utilizadas en los programas principales o subprogramas se clasifican en dos tipos:

*variables globales* y *variables locales*.

Una *variable local* es aquella que está declarada y definida dentro de un subprograma, en el sentido de que está dentro de ese subprograma y es distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. *El significado de una variable local se confina al procedimiento en el que está declarada.* Cuando otro subprograma utiliza el mismo nombre se refiere a otra posición en memoria. Se dice que tales variables son locales al subprograma en el que están declaradas.

Una *variable global* es aquella que está declarada para el programa principal, del que dependen todos los subprogramas.

La parte del programa/algoritmo en que una variable se define se conoce como *ámbito*.

El uso de variables locales tiene muchas ventajas. En particular, hace a los subprogramas independientes, con la comunicación entre el programa principal y los subprogramas manipulados estructuralmente a través de la lista de parámetros. Para utilizar un procedimiento solo necesitamos conocer lo que hace y no tenemos que estar preocupados por su diseño, es decir, cómo están programados.

Programa DEMO

Tipo X, X1, ... Ámbito de X

**.**

**.** Procedimiento A

Tipo Y, Y1, ... Ámbito de Y

**.**

**.** Procedimiento B

**.** Tipo Z, Z1, ...

**.**

**.** Ámbito de Z

**.**

Procedimiento C

Tipo W, W1, ... Ámbito de W

**.**

**.**

**.**

*Ámbito de Identificadores*



Una variable local a un subprograma no tiene ningún significado en otro subprograma. Si un

subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros programas, es decir, no pueden utilizar este valor. A veces, también es necesario que una variable tenga el mismo nombre en diferentes subprogramas.

Por el contrario, las variables globales tienen la ventaja de compartir información de diferentes subprogramas sin una correspondiente entrada en la lista de parámetros.

El *ámbito de un identificador* es la parte del programa donde se conoce el identificador. Las variables definidas en un ámbito son accesibles en el mismo, es decir, en todos los procedimientos interiores.

***2.4 Comunicación con subprogramas: Paso de parámetros***

Cuando un programa llama a un subprograma, la información se comunica a través de la lista de parámetros y se establece una correspondencia automática entre los parámetros formales y actuales. *Los parámetros actuales son sustituidos o utilizados en lugar de los parámetros formales.*

La declaración del subprograma se hace con:

**Procedimiento** nombre (clase tipo\_de\_dato: F1 clase tipo\_de\_dato: F2

...................... clase tipo\_de\_dato: Fn)

**.**

**.**

**. fin\_procedimiento**

Y la llamada al subprograma con:

**Nombre** (A1, A2, ... An);

Donde F1, F2, ... Fn son los parámetros formales y A1, A2, ... An los parámetros actuales o reales. Existen dos métodos para establecer la correspondencia de parámetros:

1. *Correspondencia posicional*. La correspondencia se establece aparejando los parámetros reales y formales según su posición en la lista: así, Fi se corresponde a Ai, donde i=1, 2, ... n. Este método se complica en legibilidad cuando el número de parámetros es muy grande.

2. *Correspondencia por el nombre explícito*. En este método, en la llamada se indica explícitamente la correspondencia entre los parámetros reales y formales. (Ada)

Por ej.:**Sub**(YgB,Xg30)

Por lo general, la mayoría de los lenguajes usan exclusivamente la correspondencia posicional, y es el que usamos nosotros en la materia.

Los métodos más empleados para realizar el paso de parámetros son: *paso por valor* y *paso por referencia*.

***2.4.1 Paso por valor***

Este método se utiliza en muchos lenguajes de programación: por ej., C, Modula-2, Pascal, Algol y Snobol. La razón de su popularidad es la analogía con los argumentos de una función, donde los valores se proporcionan en el orden de cálculo de resultados. Los parámetros se tratan como variables locales y los valores iniciales se proporcionan copiando los valores de los correspondientes argumentos.

Cualquier cambio realizado en los valores de los parámetros formales durante la ejecución del subprograma se destruyen cuando se termina el subprograma.

***2.4.2 Paso por parámetro***

En numerosas ocasiones se requiere que ciertos parámetros sirvan como parámetros de salida, es decir, se devuelvan los resultados a la unidad o programa que llama. Este método se denomina



*paso por parámetro*. La unidad que llama pasa a la unidad llamada la dirección del parámetro

actual (que está en el ámbito de la unidad llamante). Una referencia al correspondiente parámetro formal se trata como una referencia a la posición de memoria, cuya dirección se ha pasado. Entonces una variable pasada como parámetro real es compartida, es decir, se puede modificar directamente por el subprograma.

La característica de este método se debe a su simplicidad y su analogía directa con la idea de que las variables tienen una posición de memoria asignada desde la cual se pueden obtener o actualizar sus valores.

En este método los parámetros son de entrada/salida y se denominan parámetros variables.

Los parámetros valor y los parámetros variable se suelen definir en la cabecera del subprograma. En el siguiente ejemplo, en el mismo subprograma se especifican parámetros por valor y por

referencia:

**Procedimiento** Ejemplo (i: entero; **var** j: entero);

.

.

.

**Fin\_procedimiento**

Parámetro por valor

Parámetro por referencia

***2.4.3 Síntesis de la transmisión de parámetros***

Ya dijimos que los métodos más comunes de transmisión de parámetros son *por valor* y *por referencia.*

El paso de un parámetro por valor significa que el valor del argumento –*parámetro actual o real*- se asigna al parámetro formal. En otras palabras, antes de que el subprograma comience a ejecutarse, el argumento se evalúa a un valor específico (por ejemplo 8 o 12). Este valor se copia entonces en el correspondiente parámetro formal dentro del subprograma.

Una vez que el procedimiento arranca, cualquier cambio del valor de tal parámetro formal no se refleja en un cambio en el correspondiente argumento. Esto es, cuando el subprograma se termine, el argumento actual tendrá exactamente el mismo valor que cuando el subprograma comenzó, con independencia de lo que haya sucedido al parámetro formal. Estos parámetros de entrada se denominan *parámetros valor*.

El paso de un *parámetro por referencia* o *dirección* se llama *parámetro variable*. En este caso, la posición o dirección de memoria (no el valor) del argumento o parámetro actual se envía al subprograma. Si a un parámetro formal se le da el atributo de parámetro variable, entonces un cambio en el parámetro formal se refleja en un cambio en el correspondiente parámetro actual, ya que ambos tienen la misma posición de memoria.



**3**-  **Recursividad**

Un *procedimiento* o *función recursiva* es aquella que se llama a sí misma. Esta característica permite a un procedimiento recursivo repetirse con valores diferentes de parámetros. La recursión es una alternativa a la iteración.

Normalmente, una solución recursiva es menos eficiente en términos de tiempo de ejecución que una solución iterativa, debido al tiempo adicional de llamadas a procedimientos.

Pero en muchos casos, la recursión permite especificar una solución más simple y natural para resolver un problema que en otro caso sería difícil. Por ello, la recursividad es una herramienta muy potente para la resolución de problemas y la programación.

*Se dice que un objeto es recursivo si forma parte de sí mismo o se define en función de sí mismo.*

La recursión es un método general de resolver los problemas, reduciéndolos a problemas más simples de un tipo similar. La herramienta necesaria y suficiente para expresar los programas recursivamente es el *procedimiento* o *subrutina*, pues permite dar a una proposición un nombre con el cual puede ser llamado.

Si un procedimiento P contiene una referencia explícita a sí mismo, se dice que es *directamente recursivo*; si P tiene una referencia a otro procedimiento Q, que incluye una referencia (directa o indirecta) a P, entonces se dice que P es *indirectamente recursivo*.

Ej.: Sea la función F(x)= 2 F(x-1) + x2

F(0)= 0

F(1)= 2 F(0) + 12 = 1

F(2)= 2 F(1) + 22 = 6

...

función F (x:entero):entero

Si x=0 entonces

F=0;

CASO BASE: valor para el cual la función se conoce directamente sin necesidad de la recursión.

sino

F=2\*F(x-1)+x2

Llamada Recursiva

Ä Siempre debe estar definida una condición que implique la terminación.

Ä Siempre se deben tener en cuenta uno o más casos base, que se puedan resolver sin recursión.

Ä Progreso: para los casos que deben resolverse recursivamente, la llamada recursiva siempre debe tender a un caso base.

Se acostumbra asociar un conjunto de objetos locales a un procedimiento, esto es, un conjunto de variables, constantes, tipos y procedimientos que se definen localmente a este procedimiento y que carecen de existencia o significado fuera del mismo. Cada vez que ese procedimiento se activa de modo recursivo, se crea un nuevo conjunto de variables locales acotadas.

El problema principal con la recursión reside en los costos ocultos de su funcionamiento interno; aunque estos costos casi siempre son justificables, porque los programas recursivos no solo simplifican el diseño del algoritmo sino que también tienden a proporcionar un código más claro. La recursión nunca debe usarse para sustituir un simple ciclo FOR.

En la aplicación práctica es obligatorio demostrar que la profundidad final de la recursión no solo es finita, sino que en realidad es muy pequeña. Ello se debe a que, luego de cada activación recursiva de un procedimiento P, cierta cantidad de memoria se necesita para alojar sus variables (cuidado con el overflow y el underflow).



La mayoría de los ejercicios que se resuelven de manera recursiva, se pueden resolver por medio

de estructuras de control iterativas.

|  |  |
| --- | --- |
| Mientras <CONDICIÓN> hacer  <ACCIÓN> | Procedimiento P (parámetros) Si <CONDICIÓN>  entonces <ACCIÓN>  P(parámetros) |
| Repetir  <SENTENCIA>  hasta <CONDICIÓN> | Procedimiento P (parámetros)  <SENTENCIA>  Si no<CONDICIÓN>  entonces <ACCIÓN> P(parámetros) |
| Para y:=1 a n hacer  <SENTENCIA> | Procedimiento P (i) Si i<= n  entonces <SENTENCIA> P(i+1) |

Estas equivalencias no desvirtúan la naturaleza de la recursión, sino que permiten apreciar la posibilidad de prescindir de la iteración para modelar problemas resolubles mediante bucles y repeticiones.

***3.1 ¿Cuándo no usar la recursión?***

Los algoritmos recursivos son convenientes cuando el problema o los datos que deben manipularse están definidos en términos recursivos. Con todo, ello no significa que tales definiciones recursivas garantice que un algoritmo recursivo es la mejor manera de resolver el problema.

No se debe utilizar la recursión cuando la iteración ofrece una solución obvia. Sin embargo, esto no debe hacer que evitemos sistemáticamente el uso de la recursión.

La recursión se presta a muchas aplicaciones. El hecho de que los procedimientos recursivos se realicen en máquinas esencialmente no recursivas demuestra que, en la práctica, todo programa recursivo puede transformarse en uno totalmente iterativo. Sin embargo para ello se requiere el manejo explícito de una pila de recursión, y esas operaciones a menudo oscurecen la esencia de un programa, al grado que se torna difícil de entender.

Cuando nos encontramos con algoritmos de naturaleza más bien recursiva que iterativa, debemos formular los procedimientos recursivos. En los casos en que las estructuras de datos hacen obvia y natural la elección de soluciones recursivas se debe utilizar esta técnica.

La eficacia de un algoritmo recursivo viene medida por una serie de factores:

Ä *Tiempo de ejecución*

Ä *Espacio de memoria ocupado por el algoritmo*

Ä *Legibilidad y facilidad de comprensión*

Desde el punto de vista del tiempo de ejecución, los algoritmos recursivos son más lentos y ocupan mayor espacio en memoria, con el inconveniente que puede suponer en el caso de poca memoria disponible.

Sin embargo, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa; en estos casos la ganancia en claridad puede compensar con creces el coste en tiempo y en memoria de la ejecución de programas recursivos.



Las definiciones recursivas se caracterizan por su elegancia y simplicidad en contraste con la

dificultad y falta de claridad de las definiciones repetitivas.

Algunos programadores utilizan la recursividad como una herramienta conceptual, por su simplicidad y fácil comprensión; y una vez que han escrito la forma recursiva, buscan la forma de convertirla en una versión iterativa y así ganar en eficiencia, en tiempo de ejecución y ocupación de memoria.

De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).

***3.2 Algoritmo Divide y Vencerás***

Los algoritmos divide y vencerás resuelven un problema descomponiendo en varios subproblemas que son similares al problema original, excepto en que son de un tamaño más pequeño. Cada uno de los subproblemas son resueltos ahora de manera independiente. A continuación, los resultados obtenidos de la solución de estos subproblemas se combinan para producir la solución al problema original. Dado que cada uno de los subproblemas es similar al problema original, está justificado el empleo de un planteamiento recursivo.

Este método de diseñar la solución de un problema principal obteniendo las soluciones de sus subproblemas se conoce como *diseño descendente* o *top-dow*.

Los pasos seguidos en un algoritmo divide y vencerás pueden resumirse como siguen:

1) Dividir el problema original en un cierto número de subproblemas similares pero más pequeños.

2) Vencer o resolver los subproblemas. Si los subproblemas son lo suficientemente pequeños, se resuelven directamente, sino, deberían resolverse recursivamente.

3) Combinar las soluciones de los subproblemas para producir la solución final.

Un algoritmo “divide y vencerás” puede ser definido de manera recursiva, de tal modo que se llama a sí mismo aplicándose cada vez a un conjunto menor de elementos. La condición para dejar de hacer llamadas es, normalmente la obtención de un solo elemento.

El algoritmo de búsqueda binaria es un ejemplo típico de esta técnica algorítmica. La lista ordenada de elementos se divide en dos mitades de manera que el problema de búsqueda de un elemento se reduce al problema de búsqueda en una mitad; así se prosigue dividiendo el problema hasta encontrar el elemento, o bien decidir que no se encuentra.

***3.3 Algoritmo de vuelta atrás (Backtracking)***

Esta técnica algorítmica recurre a realizar una búsqueda exhaustiva sistemática de una posible solución al problema planteado. El procedimiento general es descomponer el proceso de tanteo de una solución en tareas parciales. Cada tarea parcial se expresa frecuentemente en forma recursiva.

El proceso general de los algoritmos de “vuelta atrás” se contempla como un método de prueba o búsqueda, que gradualmente construye tareas básicas y las inspecciona para determinar si conduce a la solución del problema. Si una tarea no conduce a la solución, prueba con otra tarea básica. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no hay solución por haberse agotado todas las opciones que probar.

Aplicamos esta técnica algorítmica al problema de la vuelta del caballo, que se describe a continuación:

*En un tablero de ajedrez de N x N casillas. Un caballo sigue los movimientos de las reglas del ajedrez. El caballo se sitúa en la casilla de coordenadas (X1 ,Y1). El problema consiste en encontrar, si existe, un círculo que permita al caballo pasar exactamente una vez por cada una de las casillas del tablero.*

*La tarea básica en que se va a basarse el problema es la de que el caballo realice un nuevo movimiento, o bien decidir que ya no quedan movimientos posibles. El algoritmo que*



*expondremos a continuación trata de llevar a efecto un nuevo movimiento del caballo con objeto*

*de visitar una vez todas las casillas.*

**Procedimiento Caballo**

Inicializar cuenta de opciones de selección

**Repetir**

Seleccionar nuevo movimiento hacia la solución

**Si** (Está en el tablero) y (No pasó ya) **entonces**

Anotar el movimiento

**Si** No completada tablero **entonces**

**Caballo** a partir del nuevo movimiento

**Si** No alcanza solución completa **entonces**

Borrar anotación

**Fin\_si**

**Fin\_si**

**Fin\_si**

**Hasta** (Completo tablero) o (No más posibles movimientos)

**Fin\_Caballo**

En los algoritmos de vuelta atrás siempre hay nuevas tentativas en busca de solución, nuevos ensayos. En el caso de que un ensayo no conduzca a alcanzar la solución, se da marcha atrás. Esto es, se borra la anotación hecha al realizarse el ensayo y se vuelve a hacer otro, en el caso de que sea posible (en el caso de un caballo, este puede realizar hasta ocho movimientos desde una posición dada).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **.**3 |  | **.**2 |  |
| **.**4 |  |  |  | **.**1 |
|  |  | . |  |  |
| **.**5 |  |  |  | **.**8 |
|  | **.**6 |  | **.**7 |  |

La característica principal de los algoritmos de vuelta atrás es intentar realizar pasos que se acercan cada vez más a la solución completa. Cada paso es anotado, borrándose tal anotación si se determina que no conduce a la solución, esta acción constituye una vuelta atrás. Cuando se produce una “vuelta atrás” se ensaya con otro paso (otro movimiento). En definitiva, se prueban sistemáticamente todas las opciones posibles hasta encontrar una solución, o bien agotar todas las posibilidades sin llegar a la solución.

Otros ejemplos de algoritmos con vuelta atrás son el problema de las ocho reinas, el problema del laberinto, el problema de la mochila, generación de las permutas de n elementos, los cuales están a disposición para su análisis en la bibliografía recomendada, y algunos se desarrollan en la práctica.

***3.4 Problema de la selección óptima***

En la estrategia de “vuelta atrás” se obtiene una única solución. Con el mismo criterio, se ha hecho una ampliación para encontrar así todas las posibles soluciones. Ahora no se trata de encontrar una situación fija o un valor predeterminado, sino de *encontrar del conjunto de posibles soluciones la óptima según una serie de restricciones definidas*.

En términos reales, este es *el problema del viajante de comercio que tiene que hacer las maletas seleccionando entre n artículos, aquellos cuyo valor total sea un máximo (lo óptimo, en este caso, es el máximo establecido) y su peso no exceda de una cantidad.*

Seguimos aplicando la técnica de “vuelta atrás” para generar todas las soluciones posibles, y cada vez que se alcance una solución guardarla si es mejor que las soluciones anteriores según la restricción definida.



**Procedimiento Selección Óptima**

Inicializar cuenta de opciones de selección

**Repetir**

Seleccionar nuevo paso hacia la solución

**Si** valido **entonces**

Anotar el movimiento

**Si** No completada solución **entonces**

**Selección Óptima** a partir del nuevo paso

**Si** No alcanza solución completa **entonces**

Borrar anotación

**Fin\_si**

**Sino**

**Si** Mejor(Solución) **entonces** */\*\*Compara con la mejor\*\*/*

Óptimo:= Solución */\*\* solución guardada \*\*/*

**Fin\_si**

**Fin\_si**

**Fin\_si**

**Hasta** (Completa solución) o (No más opciones)

**Fin\_Caballo**

La tarea básica en esta búsqueda sistemática es investigar si un objeto *i* es adecuado incluirlo en la selección que se irá acercando a una solución aceptable, y continuar con la búsqueda con el siguiente objeto. En el caso de que lo que haya que hacer sea excluirlo de la selección actual, el criterio para seguir con el proceso de selección actual es que el valor total todavía alcanzable después de esta exclusión no sea menor que el valor óptimo (máximo) encontrado hasta ahora. Cada tarea realiza las mismas acciones que la tarea anterior, por lo que puede expresarse recursivamente. Al estar buscando la selección óptima, hay que probar con todos los objetos del conjunto.

***3.5 Problema de los Matrimonios Estables***

El planteamiento del problema es el siguiente: dados dos conjuntos A y B, disjuntos y con igual número de elementos, *n*, hay que encontrar un conjunto de *n* pares (*a*,*b*), tales que *a* pertenece a A y *b* pertenece a B, y además cumplen con determinadas condiciones.

En el problema concreto de los matrimonios estables, A es un conjunto de hombres y B un conjunto de mujeres. A la hora de elegir pareja, cada hombre y cada mujer tienen distintas preferencias. Se trata de formar parejas, matrimonios estables. De las *n* parejas, en cuanto exista un hombre y una mujer que no formen pareja pero que se prefieran frente a sus respectivas parejas, se dice que la asignación es inestable ya que tenderá a una ruptura para buscar la preferencia común. Si no existe ninguna pareja inestable se dice que la asignación es estable.

Este tipo de planteamientos caracteriza ciertos problemas en el que hay dos conjuntos de elementos y se ha de hacer una elección según una lista de preferencias. Ej.: conjunto de ofertas de vacaciones y conjunto de turistas que quieren elegir una de ellas; conjunto de carreras y conjunto de alumnos que deben elegir una, etc.

Es como hacer la selección óptima para cada miembro de cada conjunto, comprobando que sean estables.



**4- Técnicas de diseño de algoritmos**

A continuación estudiaremos algunas clases generales de técnicas algorítmicas. Cada una de estas técnicas tiene propiedades especiales que las hacen apropiadas para resolver ciertos tipos de problemas. No se deberían aplicar estas técnicas indiscriminadamente a cualquier problema que se esté considerando. La estructura del problema es la que determina habitualmente la posibilidad de aplicar una técnica específica.

Empezamos estudiando los algoritmos de *Fuerza Bruta*. Para la mayoría de los problemas, un planteamiento de *fuerza bruta* representa la solución más simple, aunque no siempre la más eficiente. Se suele usar un algoritmo de *fuerza bruta* como base para la comparación frente a algoritmos alternativos que se desarrollan usando alguna técnica más sofisticadas de las que estudiaremos.

Fuerza bruta Algoritmo ávido Divide y vencerás

Algoritmo con retroceso Programación dinámica Algoritmos probabilísticos

***4.1 Algoritmos de fuerza bruta***

Estos algoritmos se caracterizan por una falta de sofisticación en su acercamiento a la solución. Toman la ruta más obvia o directa, sin ningún intento de minimizar el número de operaciones requeridas para calcular la solución.

La técnica consiste en plantear todo el espacio de soluciones posibles hasta encontrar una solución, o bien explorando en un orden arbitrario, o bien construyendo una solución aproximada y mejorarla poco a poco hasta obtener una solución aceptable.

Lo que caracteriza a esta clase de algoritmo es que el proceso o exploración, mejora la solución sin escoger ningún criterio de eficiencia sino buscando que la expresión del algoritmo sea sencilla la cual conduce a programas cortos y claros aunque posiblemente poco eficiente.

**\*** Planeamiento sencillo

**\*** Exploración exhaustiva

**\*** Mejoras sucesivas

**\*** Sencillez frente a eficiencia

***4.2 Algoritmos ávidos o voraces***

Los algoritmos ávidos funcionan en fases. En cada fase, se toma una decisión que parece buena, sin considerar las consecuencias futuras. En general, esto significa que se escoge algún optimo local. Esta estrategia de “tomar lo que se pueda ahora” es de donde proviene el nombre de esta clase de algoritmos. Cuando el algoritmo termina, esperamos que optimo local sea igual a optimo global. Si este es el caso, el algoritmo es correcto, si no, el algoritmo ha producido una solución subóptima. Si no se requiere la mejor respuesta absoluta, se usan a veces algoritmos ávidos simples para generar respuestas aproximadas, en vez de emplear los algoritmos complejos que suelen requerirse para generar una respuesta exacta.

Es decir, en un algoritmo ávido o voraz, en cada punto de decisión se selecciona la opción que tiene el coste inmediato más pequeño (local), sin intentar considerar si esta opción es parte de una solución óptima para el problema completo (una solución global).



El aspecto más atrayente de este tipo de algoritmos es que son simples y eficientes, ya que

generalmente se requiere de muy poco esfuerzo para calcular cada decisión local. No obstante, para problemas generales de optimización, es obvio que esta estrategia no siempre producirá soluciones óptimas globales. Sin embargo, existen ciertos problemas de optimización en los cuales una estrategia voraz o ávida garantiza de hecho la obtención de una solución globalmente óptima.

***4.3 Divide y vencerás y Algoritmos con Retroceso (backtracking)***

Ya desarrollados antes.

***4.4 Programación dinámica***

Los algoritmos diseñados usando *programación dinámica* son similares a aquellos desarrollados usando *divide y vencerás* en que ambos resuelven un problema descomponiéndolo en varios subproblemas que pueden resolverse recursívamente.

La diferencia entre los dos es que en el enfoque de *programación dinámica*, los resultados obtenidos de la resolución de subproblemas más pequeños son reutilizados en el cálculo de subproblemas más grandes. De este modo, la programación dinámica es una *técnica ascendente* que comienza habitualmente resolviendo los subproblemas más pequeños, guardando estos resultados, y entonces reutilizándolos para resolver subproblemas cada vez mayores hasta que se obtiene la solución del problema original.

Esto supone un contraste con respecto al planteamiento *divide y vencerás*, que resuelve los problemas de una manera *descendente*. En este caso el problema original se resuelve descomponiéndolo en subproblemas cada vez más pequeños, pero no se hace ningún intento para reutilizar los resultados previos en la solución de ninguno de los subproblemas.

Es importante darse cuenta de que un planteamiento de programación dinámica solo está justificado si existe cierto grado de solapamiento en los subprogramas, para evitar tener que calcular el mismo resultado dos veces. Esto se consigue habitualmente construyendo tablas en memoria, y rellenándolo con los resultados obtenidos conforme se van calculando. Estos resultados son usados entonces para resolver subproblemas más grandes.

La programación dinámica se utiliza a menudo para resolver problemas de optimización, en el cual habitualmente hay un gran número de posibles soluciones, y cada una tiene un coste asociado. El objetivo es encontrar la solución óptima.

***4.5 Algoritmos probabilistas***

El comportamiento de un *algoritmo probabilista* depende no solo de los datos de entrada sino también de los valores producidos por un generador de números aleatorios. Si alguna parte de un algoritmo supone el escoger entre algún número de alternativas, y es difícil determinar la opción óptima, entonces a menudo es más eficiente escoger un curso de acción al azar más que gastar tiempo en determinar la mejor alternativa. Esto es particularmente cierto en los casos en los que hay un gran número de opciones, la mayoría de las cuales son buenas.

Aunque hacer probabilista un algoritmo no mejorará su tiempo de ejecución en el peor caso, puede usarse para asegurar que ninguna entrada particular provoque *siempre* el comportamiento del peor caso. Concretamente, como el comportamiento de un algoritmo probabilista se determina por una secuencia de números aleatorios, sería inusual para el algoritmo comportarse de la misma forma en ejecuciones sucesivas, incluso cuando se le suministren los mismos datos de entrada.



**5- Bibliografía**

***Nombre del libro***

***Autor***

***Editorial***

***Capítulos del libro que se tubo en cuenta para armar el apunte de esta unidad***

(1) “Estructura de Datos y Algoritmos en JAVA”

Autores: Michael T. Goodrich y Roberto Tamassia

Editorial: CECSA Capítulo: 3

(2) “Análisis y Diseño de Algoritmos: un enfoque teórico y práctico” Autor: José Ignacio Pelaez Sanchez

Editorial: Universidad de Málaga / Manuales

Capítulos: 2, 4, 5, 6, 7, 8 y 9

(3) “Estructura de Datos. Algoritmos, abstracción y objetos” Autor: Luis Joyanes Aguilar

Editorial: McGraw Hill

Capítulos: 1, 9, 16

(4) “Fundamentos de Programación: Algoritmos y Estructuras de datos” Autor: Luis Joyanes Aguilar

Editorial: McGraw Hill

Capítulos: 5

(5) “Estructura de Datos y Algoritmos” Autor: Mark Allen Weiss

Editorial: Addison-Wesley Iberoamericana

Capítulos: 2 y 10

(6) “Estructuras de datos y algoritmos” Autor: Alfred V. Aho

Editorial: Addison-Wesley Iberoamericana

Capítulos: 1, 9 y 10

(7) “Estructura de Datos, Algoritmos y programación orientada a objetos” Autor: Gregory Heileman

Editorial: McGraw Hill

Capítulos: 2

(8) “Estructura de Datos”

Autor: Osvaldo Cairó y Silvia Guardati Buemo

Editorial: McGraw Hill

Capítulos: 4

Sitios Web Recomendados:

<http://es.gotdotnet.com/quickstart/howto/>

Tutorial de tareas comunes de Microsoft: está diseñado para ayudar a encontrar rápidamente respuestas a las preguntas de programación más frecuentes. <http://www.microsoft.com/spanish/msdn/default.asp> <http://www.microsoft.com/spanish/msdn/Argentina/default.asp>

Sitio de Microsoft con todo lo que necesitan de herramientas de esta empresa, cursos gratuitos, etc..