CAPÍTULO 7

ALGORITMOS Y SU COMPLEJIDAD

La tarea de programación está ligada al objetivo de obtener algoritmos que resuelvan un problema con la mayor eficiencia posible; de hecho es sorprendente comprobar las múltiples formas como podemos resolver un mismo problema y las ventajas que conseguimos, en términos de eficiencia, al buscar soluciones alternativas a las ya conocidas o consideradas como evidentes.

El objetivo de este capítulo es doble: presentar la variedad de algoritmos existentes para resolver una serie, necesariamente reducida, de problemas habituales de distinta naturaleza y, en paralelo, formalizar la idea de "mayor eficiencia", estableciendo el concepto de *complejidad algorítmica* que desde el punto de vista computacional es básico. Esto nos permite introducir determinados conceptos relacionados con los problemas que son potencialmente resolubles.

7.1. MEDIDA DE LA EFICIENCIA Y DE LA COMPLEJIDAD ALGORITMICA

Hasta ahora, el concepto de mayor eficiencia de un algoritmo frente a otro lo hemos introducido de forma intuitiva, lo que no ha sido óbice para apostar por algún método para obtener algoritmos más eficientes (p.e. divide y vencerás).

Para comparar y analizar la eficiencia de los algoritmos, éstos los consideraremos escritos en un lenguaje de programación de alto nivel, pero aún empleando la misma representación, establecer una medida precisa de la eficiencia de un algoritmo no es fácil. En efecto, fijémonos en que una definición de eficiencia podría ser el número de instrucciones que tiene el programa; sin embargo esto no se correspondería, con el concepto intuitivo que tenemos de eficiencia (según el cual, el algoritmo más eficiente sería aquel que tardase menos tiempo en resolver el problema sobre una misma máquina), dado que todas las instrucciones no utilizan el mismo tiempo de procesador aun realizando la misma función (Ver ejemplo de la Figura 7.1).

desde i:=1 **hasta** 10000	**desde** i:=1 **hasta** 10000

x ← i^2 **fin-desde**	x ← i*i **fin-desde**

Fig. 7.1. *El primer algoritmo es más costoso que el segundo, a pesar de hacer la misma función, a causa de que la potencia es mucho más lenta que el producto.*

Es más, el número de instrucciones puede reducirse sin que por ello disminuya el número de instrucciones que realmente deben procesarse; así por ejemplo, un bucle lo podemos programar tanto con una estructura **desde-hasta** como con una estructura **mientras** y aunque en este último caso se necesitan más instrucciones (Ver Figura 7.2), realmente el tiempo de proceso es el mismo[1].

desde i:=1 **hasta** 1000 acciones S **fin-desde**	i ← 1 **mientras** i <= 1000 acciones S i ← i+1 **fin_mientras**

Fig. 7.2. *Los dos bucles implican los mismos cálculos, con independencia del numero de instrucciones de sus estructuras.*

Por otra parte, como ya hemos hecho notar en capítulos anteriores, el propio concepto de *tipo de dato* se utiliza, entre otras cosas, para emplear los métodos más adecuados para realizar las operaciones entre los datos. Así, para un mismo procesador, efectuar el producto de dos números es mucho más lento si estos números son de tipo Real que si son de tipo Entero, aun siendo los mismos números; sin embargo ambos casos son similares desde el punto de vista algorítmico.

Todo lo anterior, unido a la dificultad de dar una medida de tiempo relativo ligada a la ejecución de cada instrucción (sobre todo en lo que al acceso a periféricos se refiere) hace que una definición precisa que se corresponda con el concepto intuitivo de eficiencia tenga poco sentido. Cuestiones como las comentadas anteriormente, el bucle *desde-hasta* o el tipo de dato más adecuado, están ligadas a cuestiones de implementación, más que al propio diseño del algoritmo, y pueden optimizarse con suficiente práctica en la codificación de algoritmos. Sin embargo, existen elementos ligados al propio diseño del algoritmo, que potencialmente son mucho más impactantes sobre el tiempo de cálculo, que las consideraciones anteriores, si el programador no considera adecuadamente la situación. Como ejemplo analicemos los algoritmos de la Figura 7.3, que en principio, calculan el valor de x^{2^n} y veamos su comportamiento en función del valor de n.

m←x	m←1

[1] Si bien es cierto que el bucle *desde-hasta* es ligeramente más rápido que el bucle *mientras*.

desde j:=1 **hasta** n, **hacer** m←m*m **fin-desde**	**desde** j:=1 **hasta** 2^n , **hacer** m←m*x **fin-desde**

Fig. 7.3. *Ejemplo de dos algoritmos que calculan* x^{2^n}

En el primer algoritmo se necesitan realizar n multiplicaciones, mientras que en el segundo se realizan 2^n operaciones. Esto significa que para n=100 en un caso realizamos 100 multiplicaciones y en el otro caso $2^{100} = 1.26*10^{30}$. Esto significa que disponiendo de un computador que efectuara un millón de multiplicaciones por segundo (lo cual implica que procesaría varios millones de instrucciones de lenguaje máquina por segundo), para ejecutar el primer algoritmo empleará 0.0001 seg. (una diezmilésima de segundo) y para el otro $1{,}26*10^{24}$ seg (¡aproximadamente $4*10^{16}$ años!), o sea que si hubiese empezado a calcular centenas de miles de millones de años antes de que existiese el Universo, aún no habría acabado). Este ejemplo nos da elementos para poder avanzar en el concepto de complejidad de un algoritmo, ya que esta definición no debe necesariamente discriminar entre la eficiencia de dos algoritmos cuyo tiempo de computación sea del mismo "orden de magnitud", pero que sí que lo debe hacer entre algoritmos como los expuestos en la Figura 7.3.

En ambos algoritmos, que resuelven el mismo problema (calcular x^{2^n}) podemos ver que el número de operaciones a realizar dependerá del valor de n; se dice que n es la talla o tamaño del problema. Si, por poner otro ejemplo, el caso fuera el de buscar un valor en un vector, el tamaño del problema sería el número de elementos del vector, ya que, sea cual sea el algoritmo, es obvio que el número de operaciones dependerá del número de elementos del mismo. En general vemos que cada problema tiene un tamaño (que determinaremos en cada caso) y que el coste del algoritmo que resuelva el problema dependerá de dicho tamaño.

7.1.1 ORDEN DE COMPLEJIDAD

La complejidad de un algoritmo deberá estar relacionada con el número de operaciones elementales necesarias (asignaciones, comparaciones, sumas, restas, multiplicaciones, divisiones, etc.) para resolver un problema dado. El número de estas operaciones dependerá de los datos de entrada y éstos pueden dar lugar desde una solución especialmente favorable hasta una especialmente costosa. Por ejemplo en la búsqueda secuencial de un vector ordenado, si el valor buscado es el menor lo obtendremos con muy pocas operaciones, sin embargo si buscamos el último, la solución supondrá recorrer todo el vector. Por ello se habla del mejor, del peor caso y del caso medio. Para el peor caso de un algoritmo, expresaremos la función que mide el numero de operaciones según el tamaño del problema como f(n), que tomaremos como medida aproximada de la eficiencia.

Definición: Diremos que la **complejidad del algoritmo es del orden de** g(n) (lo representaremos por $O(g(n))$) si existe un n_0 tal que, para todo valor de $n \geq n_0$, existe una constante C tal que $|f(n)| \leq C\ |g(n)|$. Con esta definición, si f(n) es un polinomio de grado k, el algoritmo correspondiente tendrá una complejidad $O(n^k)$, independientemente de cuales sean los coeficientes y términos de menor grado del polinomio. De esta forma se formaliza la idea de que dos algoritmos de distinta eficiencia tengan la misma complejidad. En el caso de la figura 7.3, la complejidad de los algoritmos es de $O(n)$ y $O(2^n)$ respectivamente.

Si un algoritmo tiene una complejidad de $O(n^k)$, se dice que es **polinomial**[2] (en el caso de k=1 se le llama **lineal**, para k=2 **cuadrático** y para k=3 **cúbico**); el resto de algoritmos se conocen como **no-polinomiales**. La siguiente tabla muestra el tiempo que tardaría un computador en realizar f(n) operaciones, para distintas funciones f y distintos valores de n, suponiendo que realizase un millón de operaciones por segundo.

Tabla 7.1. *Tiempo en segundos (salvo indicación contraria) que tardan en realizarse f(n) operaciones, haciendo un millón por segundo*

f(n) \ n	10	20	30	40	50	70	100
n	0.00001 s	0.00002 s	0.00003 s	0.00004 s	0.00005 s	0.00007 s	0.0001 s
n log(n)	0.00003 s	0.00008 s	0.00014 s	0.00021 s	0.00028 s	0.00049 s	0.0006 s
n^2	0.0001 s	0.0004 s	0.0009 s	0.0016 s	0.0025 s	0.0049 s	0.01 s
n^3	0.001 s	0.008 s	0.027 s	0.064 s	0.125 s	0.343 s	1 s
n^4	0.01 s	0.16 s	0.81 s	2.56 s	6.25 s	24 s	1.6 min
n^5	0.1 s	3.19 s	24.3 s	1.7 s	5.2 min	28 min	2.7 horas
$n^{Log(n)}$	0.002 s	4.1 s	17.7 s	5 min 35 s	1 h 4min	2.3 días	224 días
2^n	0.001 s	1.04 s	17 min	12 dias	35.6 a.	37 Ma	2.6 MEU
3^n	0.059 s	58 min	6.5 a.	385495 a.	22735 Ma	52000 Ma	10^{18} MEU
n!	3.6 s	77054 a.	564 MEU	$1.6\ 10^{18}$ME	$6\ 10^{34}$	$2.4\ 10^{70}$M	2
n^n	2.7 horas	219 EU	$4.2\ 10^{14}$ME	$2.4\ 10^{28}$ME	$1.8\ 10^{67}$ME	$3\ 10^{111}$ M	$2\ 10^{182}$ M

s: segundos min: minutos h: horas a: añosMa: millones de años
EU: edad del universo MEU: millones de veces la edad del universo

Comparando las casillas inferiores de esta tabla (sombreadas) con el resto, se hace evidente la importancia de estimar la complejidad de los algoritmos: aquellos cuya complejidad del orden de las últimas filas de la tabla (no-polinomiales) no resultan útiles más que para tamaños muy reducidos del problema a resolver, y por tanto con muy reducida aplicación práctica.

[2] También conocidos como Algoritmos de tiempo polinomial.

7.1.2 ANÁLISIS DE COMPLEJIDAD DE LOS ALGORITMOS DE BÚSQUEDA

Vamos ahora a comparar la complejidad de dos algoritmos, ya estudiados, que resuelven un mismo problema: búsqueda secuencial y binaria. Para ponernos en condiciones de partida semejantes, supongamos que trabajamos con un vector ordenado de n elementos (única situación en la que es posible, aplicar la búsqueda binaria).

En el caso de la búsqueda secuencial, el peor caso se dará cuando el elemento buscado no exista en el vector o cuando éste esté situado en la ultima posición, por lo que el numero de comparaciones, será igual al tamaño del vector (en este caso el número de elementos del vector, n, es el tamaño del problema), por consiguiente, O(n). (Obsérvese que el caso medio, supondrá aproximadamente (n+1)/2 comparaciones, con lo cual seguimos teniendo un problema de complejidad lineal).

Para analizar la búsqueda binaria usaremos también el número de comparaciones que necesitamos, en el peor caso, que se dará cuando no encontremos el elemento buscado o este se halle en uno de los extremos del vector. Para darnos una idea, consideremos el siguiente vector: 1, 2, 3, 4, 5, 6, 7. Si buscamos el valor 7, tendremos que efectuar 3 comparaciones. En realidad podemos redondear y suponer que trabajamos con la potencia de dos más próxima por exceso a n, para estimar el número máximo de subdivisiones que necesitamos en el peor caso. En nuestro caso podemos suponer que el vector tuviera 8 elementos y observar como: $3 = \log_2 8$, nos da el máximo numero de subdivisiones que tendríamos que llegar a efectuar para localizar cualquier valor del vector. Sin mucha dificultad podemos aceptar que este algoritmo presenta una complejidad de $O(\log_2 n)$, cuya gran eficiencia ni siquiera se ha representado en la tabla 7.1, ya que el número de operaciones, para el caso algorítmico, es realmente muy baja y casi despreciable en término de tiempo para un computador moderno. Así para un tamaño del vector de 100, solo serian necesarias 7 comparaciones y para el caso de n = 1000000, con 20 serían suficientes. Con ello concretamos la mayor eficiencia, ya adelantada, de un algoritmo frente a otro en el caso de vectores ordenados.

7.2. ALGORITMOS DE ORDENACIÓN Y SU COMPLEJIDAD

La ordenación o clasificación es el proceso de organizar los datos individuales de un vector en algún orden o secuencia específica (creciente o decreciente para datos numéricos o alfabéticamente para los datos tipo carácter). Durante la descripción y análisis del proceso de búsqueda, hemos tenido ocasión de ver la importancia que tiene el poder trabajar con datos ordenados; por ello no debe

extrañar que este problema ocupe un lugar especial en un curso de programación. Nótese incluso que de esta operación viene el término "ordenador".

Vamos a distinguir dos tipos de ordenación, la llamada ordenación **interna** que exige que la colección completa de elementos a ordenar estén almacenados en memoria principal (normalmente en forma de un vector) y la ordenación **externa** en la que se trabaja con la restricción de que la mayoría de datos se encuentran fuera de la memoria principal, esto es, almacenados en memoria secundaria (disco o cinta). La razón para esta distinción reside en el hecho de que en un archivo es muy frecuente que existan más registros a ordenar que los que caben simultáneamente en la memoria principal; en consecuencia las hipótesis de partida son radicalmente distintas. Afortunadamente la mayoría de sistemas operativos actuales disponen de órdenes capaces de llevar a cabo la ordenación de un archivo (p.e. la orden SORT en el DOS de Microsoft). A continuación vamos a estudiar algunos algoritmos utilizados en ordenación interna de vectores.

7.2.1 ORDENACIÓN POR INSERCIÓN

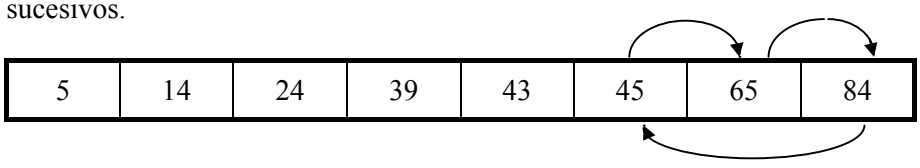
Este algoritmo consiste en insertar un elemento en el vector, en una parte ya ordenada del mismo y comenzar de nuevo con los elementos restantes; fijarse que de hecho este método es el que utilizamos cuando ordenamos un mazo de naipes por lo se le conoce también por *método de la baraja.*

Ejemplo 1:

Consideremos el siguiente vector, ya parcialmente ordenado en sus primeros siete elementos y hagamos la inserción del octavo (45) en su lugar correspondiente:

5	14	24	39	43	65	84	45

Para insertar el elemento 45, que es el siguiente en la ordenación, habrá que insertarlo entre 43 y 65, lo que supone desplazar a la derecha todos aquellas números de valor superior a 45, es decir, saltar sobre 65 y 84. Como puede observarse, el algoritmo se ejecutara en base a comparaciones y desplazamientos sucesivos.



El algoritmo de ordenación por inserción de un vector X de *N* elementos, se realiza con un recorrido de todo el vector (desde el segundo al *n*-ésimo elemento, pues un vector de un solo elemento, siempre está ordenado) y la inserción del elemento i-esimo correspondiente en el lugar adecuado, suponiendo que en cada momento el subvector X(1), X(i-1) esta ordenado En un primer esbozo del diseño, podemos escribir:

desde i := 2 **hasta** N **hacer**
seleccionar el elemento más pequeño de X(i)...X(N) e intercambiarlo con X(i)
fin_desde

Esta acción -*insertar*- que se repite y constituye el cuerpo del bucle es una operación que ya hemos tratado en vectores, pudiendo llevarse a cabo de diferentes formas; Veamos el algoritmo correspondiente y observemos como se utiliza el hecho, de que en cada paso (J) del bucle **desde**, al estar ordenado el subvector correspondiente, X(1),....., X(J-1), el valor X(J-1) es el mayor de ellos. El objetivo de cada pasada del bucle, es encontrar el valor que debe ocupar la posición j-ésima en el vector ordenado.

```
algoritmo ord_inserción
inicio
desde J = 2 hasta N hacer
AUX ← X(J)
K ← J - 1
      mientras AUX < X(K) hacer
      X(K + 1) ← X(K)
      K ←  K - 1
      fin_mientras
X(K + 1) ← AUX
fin_desde
fin
```

Para estimar la complejidad de este algoritmo, tomaremos como operaciones elementales para calcular f(n), las comparaciones y asignaciones que se realizan involucrando números a ordenar. El peor caso para obtener una ordenación ascendente del vector, se produce cuando los números están inicialmente ordenados de mayor a menor, es decir totalmente al revés. En este caso, el número de comparaciones necesarias es, en función del contador j del algoritmo:

	j=2		j=3		j=4 .	..		j=n-1		j=n
NºComparaciones:	1	+	2	+	3	+ ... +		n-2	+	n-1 = n(n-1)/2

Num. Asignaciones:

Bucle Desde : 2 + 2 + 2 ... + 2 + 2 =2(n-1)
Bucle Mientras: 1 + 2 + 3 + ... + n-2 + n-1 = n(n-1)/2

Así, $f(n) = \frac{n^2}{2} - \frac{n}{2} + 2n - 2 + \frac{n^2}{2} - \frac{n}{2} = n^2 + n - 2$ y por tanto el algoritmo es de complejidad $O(n^2)$.

Nótese que en el mejor caso (números inicialmente en el orden correcto), el número de operaciones elementales es sensiblemente inferior (3n-3). En efecto:

	j=2	j=3	j=4		j=n-1	j=n
Num. Comparaciones:	1 +	1 +	1 +	... +	1 +	1 = n-1
Num. Asignaciones						
Bucle Para :	2 +	2 +	2 +	... +	2 +	2 =2(n-1)
Bucle Mientras:	0 +	0 +	0 +	... +	0 +	0 = 0

7.2.2 ORDENACIÓN POR INTERCAMBIO

El algoritmo de clasificación u ordenación por *intercambio* se basa en el principio de comparar pares de elementos adyacentes y en caso de contravenir el orden esperado, intercambiarlos, realizando este proceso las veces necesarias hasta que estén todos ordenados.

Ejemplo 2:

Supongamos que se desea clasificar en orden descendente por el método de intercambio el siguiente vector:

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)
50	15	56	14	35	1	12	9

Los pasos a dar son:

1.- Comparar A(1) y A(2); si están en orden, se mantienen como están; en caso contrario se intercambian entre sí. Al buscar un orden descendente, no efectuamos ningún cambio pues 50 > 15 (orden descendente correcto).
2.- A continuación se comparan los elementos A(2) y A(3); de nuevo se intercambian si es necesario. En este caso 15 > 56, luego debemos intercambiarlos.
3.- El proceso continúa hasta que cada elemento del vector ha sido comparado con sus elementos adyacentes y se han realizado los intercambios necesarios.

Los sucesivos contenidos del vector a lo largo de estos pasos, son los siguientes:

	v.inicial	1ªcomp.	2ªcomp.	3ªcomp	4ªcomp	5ªcomp	6ªcomp	7ªcomp
A(8)	9	9	9	9	9	9	9	1
A(7)	12	12	12	12	12	12	1	9
A(6)	1	1	1	1	1	1	12	12
A(5)	35	35	35	35	14	14	14	14
A(4)	14	14	14	14	35	35	35	35
A(3)	56	56	15	15	15	15	15	15
A(2)	15	15	56	56	56	56	56	56
A(1)	50	50	50	50	50	50	50	50

Como se observa, al finalizar estos pasos, el elemento, cuyo valor sea menor, 1 en nuestro ejemplo, acabará ocupando la última posición; utilizando un símil físico, podemos decir que como elemento más ligero, sube hacia arriba, al igual que las burbujas de aire en un depósito o botella de agua. Por esta razón este algoritmo se conoce también como método de la burbuja.

Por tanto, en el caso de una ordenación descendente, a partir de un vector cualquiera de N elementos, tras realizar un recorrido completo por todo el vector, conseguimos que el menor de ellos, quede situado en la última posición, que es la que le corresponderá, cuando el vector quede ordenado. En el segundo recorrido, sólo será necesario abarcar a los N-1 elementos restantes, y en él el segundo valor más pequeño llegará a la penúltima posición. Siguiendo de esta manera acabaremos ordenando el vector, colocando en cada iteración, secuencialmente, un valor en su posición definitiva.

Basándonos en la anterior, podemos proponer un algoritmo de ordenación basado en dos etapas iterativas, una anidada dentro de la otra:

1ª etapa Dado un vector de N elementos Hacer el recorrido de la burbuja (ver segunda etapa) primero para N elementos, luego para el subvector con los N-1 primeros, luego para N-2, etc., hasta 2 elementos.

2ª etapa Recorrido de la burbuja: para los elementos a recorrer, comparar cada uno con el siguiente e intercambiarlos si contravienen el orden requerido.

Mostrar la transformación del vector del ejemplo 2 de ordenación descendente, tras dos ejecuciones del bucle interno (etapas o recorridos de la burbuja) del método de intercambio[3]:

[3]En el ejemplo podemos apreciar que tras dos iteraciones de la etapa externa el vector ya está ordenado y no haría falta continuar con más recorridos, con el consiguiente ahorro de tiempo. La observación de este hecho permitiría escribir un algoritmo de menor coste en el caso medio, aunque no en el peor caso..

estado inicial	después de 1ª etapa	después de 2ª etapa
9	1	1
12	9	9
1	12	12
35	14	14
14	35	15
56	15	35
15	56	50
50	50	56

La etapa interna, en un primer diseño sería:

```
desde I=1 hasta R-1 hacer {siendo R el número de elementos a recorrer}
      si A(I) < A(I+1) entonces intercambiar A(I) con A(I+1)
fin_desde
```

La acción *intercambiar* los valores de dos elementos A(I), A(I+1), es muy común y al objeto de que no se pierda ningún valor, hay que reseñar que debe recurrirse a una variable auxiliar, AUX, de forma que el procedimiento sería:

```
AUX← A(I)
A(I)← A(I+1)
A(I+1)← AUX
```

Estamos en condiciones de presentar el algoritmo completo, donde cada una de las etapas viene representada por sendos bucles anidados. En esta ocasión, una vez justificado el nombre de burbuja, vamos a ordenar de forma ascendente, esto es de menor a mayor (la forma más habitual).

```
algoritmo burbuja      {ordenación ascendente}
   inicio
     desde I=1 hasta N hacer     { lectura del vector}
       leer X(I)
     fin-desde
     desde I=1 hasta N-1 hacer
       desde J=1 hasta N-I hacer    {no utilizamos los elementos ya ordenados}
         si X(J) > X(J+1) entonces          {intercambiar}
           AUX← X(J)
           X(J) ←X(J+1)
```

La inclusión de la mejora correspondiente no se contempla en este texto y queda como un ejercicio para el lector

```
        X(J+1) ←AUX
      fin-si
    fin-desde
  fin-desde
  desde J = 1 hasta N hacer  {imprimir vector ordenado}
    escribir X(J)
  fin-desde
fin
```

Para analizar la complejidad del algoritmo, vamos a considerar el número de comparaciones que efectuamos. Como puede observarse el número de comparaciones será el mismo para cualquier vector, sólo las asignaciones cambiarán siendo en el peor caso tres por cada comparación, con lo que serán del mismo orden de magnitud. Este método de ordenación, para un vector de n elementos, precisa el siguiente numero de comparaciones (observando los valores que toman los contadores I y J respectivamente):

$$(1 + 2 + \quad + n\text{-}1) = (n - 1) * (n/2)$$

Por tanto, estamos, de nuevo, con un algoritmo cuadrático, por lo que es interesante que mejoremos su eficacia.

Ejemplo 3:

Describir los diferentes pasos para clasificar en orden ascendente el vector siguiente por el método de la burbuja.

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)
72	64	50	23	84	18	37	99	45	8

Las sucesivas operaciones en cada uno de los pasos necesarios hasta obtener la clasificación final se muestra en la siguiente tabla, donde aparecen sombreados los elementos que ya quedan ordenados y por tanto ya no se recorren:

	vector inicial	**1 rec. i=1**	**2 rec. i=2**	**3 rec. i=3**	**i=4**	**i=5**	**i=6**	**i=7**	**i=8**	**Fin de ordenación**
A(1)	72	64	50	23	23	18	18	18	18	8
A(2)	64	50	23	50	18	23	23	23	8	18
A(3)	50	23	64	18	37	37	37	8	23	23
A(4)	23	72	18	37	50	49	8	37	37	37
A(5)	84	18	37	64	45	8	41	41	41	41
A(6)	18	37	72	45	8	50	50	50	50	50

A(7)	37	84	45	8	64	64	64	64	64	64
A(8)	99	45	8	72	72	72	72	72	72	72
A(9)	45	8	84	84	84	84	84	84	84	84
A(10)	8	99	99	99	99	99	99	99	99	99

7.2.3 ALGORITMO DE SHELL

Tanto el método de inserción como de la burbuja son de orden cuadrático y en consecuencia su tiempo de ejecución se dispara cuando el número de elementos a ordenar es grande. Con el objetivo de mejorar ambas aproximaciones Donald Shell propuso un nuevo método que lleva su nombre y que, sin mejorar la complejidad en el peor caso, se comporta de forma sensiblemente más eficaz en la gran mayoría de casos.

Shell cayó en la cuenta que en los métodos anteriores, el comparar cada elemento con su contiguo supone, en promedio, ejecutar muchas comparaciones antes de colocar los elementos extremos en su lugar definitivo. Su propuesta consiste en modificar los saltos contiguos resultantes de las comparaciones, por saltos de mayor tamaño y con ello conseguir una ordenación más rápida. El método se basa en fijar el tamaño de los saltos constantes, pero de más de una posición.

Supongamos un vector de elementos

4	12	16	24	36	3

en el método de inserción directa, los saltos se hacen de una posición en una posición y se necesitarán 5 comparaciones. En el método de Shell, si los saltos son de dos posiciones, se realizarán tres comparaciones.

El método de Shell se basa en tomar como salto inicial el valor N/2 (siendo N el número de elementos) y luego se va reduciendo a la mitad en cada repetición hasta que el salto o distancia se reduce a 1. Por tanto habrá que manejar la variable salto, de forma que los recorridos no serán los mismos en cada iteración.

Sea un vector: X(1), X(2), X(3),..., X(N). El primer salto a dar que tendrá un valor de:

$$\frac{1+N}{2}$$

por lo que para redondear, se tomará la parte entera $\frac{1+N}{2}$

E = ent((1 + N)/2)

El algoritmo resultante será:

```
subalgoritmo Shell
inicio
   E ← N+1
   mientras E > 1
      E ← ent(E/2)
      repetir
         ID ← verdadero
         I ← 1
         desde  I = 1 hasta N-E
                si X(I) > X(I+E) entonces
                   AUX ← X(I)
                   X(I) ← X(I+E)
                   X(I+E) ← AUX
                   ID ← falso
               fin-si
         fin-desde
      hasta-que (ID = verdadero)
   fin-mientras
fin
```

Ejemplo 4:

Deducir las secuencias parciales de clasificación por el método de Shell para ordenar de modo ascendente el vector siguiente

6	1	5	2	3	4	0

Estas son:

Nº de Recorrido	*Salto*	*vector resultante*	*intercambios*
vector inicial		6,1,5,2,3,4,0	
1	3	2,1,4,0,3,5,6	(6,2),(5,4),(6,0)
2	3	0,1,4,2,3,5,6	(2,0)
3	3	0,1,4,2,3,5,6	ninguno
4	1	0,1,2,3,4,5,6	(4,2),(4,3)
5	1	0,1,2,3,4,5,6	ninguno

Aunque el análisis de la complejidad (que no haremos con detalle), nos demostraría que para el peor caso, el algoritmo de Shell es cuadrático y no mejoraría los algoritmos de ordenación anteriores (seguimos trabajando con dos bucles anidados), como hemos dicho, en la práctica se puede comprobar que la modificación propuesta por Shell supone una mejora importante para vectores no especialmente desordenados.

7.2.4 ALGORITMO DE ORDENACIÓN RÁPIDA ("QUICKSORT")

Este nuevo método, de naturaleza recursiva, llamado ordenación rápida (quicksort) propuesto por Hoare se basa en el hecho práctico de que es más rápido y fácil de ordenar dos listas pequeñas que una lista grande. Se denomina de ordenación rápida porque, en general, puede ordenar una lista de datos mucho más rápidamente que cualquiera de los métodos de ordenación ya estudiados; de hecho, se puede demostrar que su complejidad para el caso medio es $O(n* \log(n))$.

El método es un ejemplo de la estrategia "*divide y vencerás"* de forma que el vector inicial se divide en dos subvectores: uno con todos los valores, menores o iguales a un cierto valor específico, llamado *pivote*, y otro con todos los valores mayores que él. El valor elegido puede ser cualquier valor arbitrario dentro del vector. El primer paso consiste en seleccionar el valor pivote y en dividir el vector original V en dos partes:

- El subvector VI que sólo contiene valores inferiores o iguales al valor pivote
- El subvector VD que sólo contiene valores superior o iguales al valor pivote

Notemos que los subvectores VI y VD no estarán ordenados, excepto en el caso de reducirse a un único elemento (obviamente ordenado), circunstancia ésta que será especialmente útil.

Ejemplo 5:

Consideremos el siguiente vector y llevemos a cabo la anterior descomposición de valores.

18 11 27 13 9 4 16 15 25

Se elige un valor pivote, por ejemplo 13. Se recorre la lista desde el extremo izquierdo y se busca un elemento mayor que 13 (el primero es 18). A continuación, se busca desde el extremo derecho un valor menor que 13 (se encuentra el 4). Se intercambian estos dos valores y se obtiene:

4 11 27 13 9 18 16 15 25

Se sigue recorriendo el vector por la izquierda y se localiza el 27, y a continuación se busca por la derecha otro valor menor que el pivote, se encuentra a la derecha (el 9); al intercambiar estos dos valores se obtiene:

4 11 9 13 27 18 16 15 25

Al intentar este proceso una vez más, nos encontramos con que la búsqueda desde la izquierda de valores mayores que el pivote como la búsqueda desde la derecha de valores menores que el pivote, acaban cruzándose, esto es, las exploraciones desde los dos extremos acaban sin encontrar ningún valor que esté "fuera de lugar". Por tanto, lo que tenemos es que todos los valores a la derecha son mayores que todos los valores a la izquierda del pivote. Por tanto, el objetivo se ha alcanzado, ya que tenemos una partición del vector original en dos listas más pequeñas:

4 11 9 [13] 27 18 16 15 25

Obsérvese que nadie nos garantiza que la situación anterior, donde el número de valores por encima del pivote a su izquierda ha coincidido con el número de valores menores que él a su derecha, se produzca para cualquier vector. Por ello el algoritmo tiene que incluir algún mecanismo para que el propio pivote se sitúe en su posición correcta.

Siguiendo con el ejemplo anterior, hemos conseguido que el elemento pivote tenga valores menores delante y valores mayores detrás. Sin embargo, nos encontramos con que ninguna de ambas listas a la izquierda y a la derecha están ordenadas; sin embargo, basados en los resultados de la primera partición, se puede ordenar ahora cada una de las dos partes de forma independiente. Esto es, debemos ordenar los subvectores

4 11 9
27 18 16 15 25

para acabar con el vector totalmente ordenado.

Para un caso totalmente general, vamos a esbozar en primer lugar el método para conseguir una partición del vector en VI -VD:

establecer como pivote X el valor de un elemento arbitrario de la lista
mientras la partición no se termina **hacer**
recorrer desde la izquierda y buscar un valor >= X

```
        recorrer desde la derecha y buscar un valor <= X
            si los valores están desordenados entonces intercambiar sus valores
        fin-mientras
```

Si el vector es A, se deben considerar las sucesivas particiones que se van generando, A(L),..., A(R), donde L y R representan los índices de los extremos izquierdo y derecho del subvector que se está considerando. Para realizar el recorrido de las sublistas se consideran las variables índices *i* y *j*. Normalmente se supone que el valor pivote es conocido, sino es así, a falta de mejor criterio, seleccionaremos como pivote el valor central del vector.

El siguiente algoritmo efectúa la partición de un vector A(L),...,A(R), basándose en la elección de un pivote cuyo valor es X.

```
algoritmo partición
inicio
   i ← L
   j ← R
   X← A((L+R)/2) {o cualquier otra elección de pivote}
   mientras i <= j hacer
      mientras A(i) < X hacer
        i ← i + 1
      fin-mientras
      mientras A(j) > X hacer
        j ← j - 1
      fin-mientras
      si i <= j
        entonces
          AUX ← A(i)
          A(i) ← A(j)
          A(j) ← AUX
          i ← i + 1
          j ← j - 1
      fin-si
   fin-mientras
fin
```

Obsérvese, que una vez ejecutado el algoritmo de partición, obtenemos dos subvectores:

A(L) ... A(j) y A(i) ... A(R)

que será preciso ordenar a continuación y por separado para ser unidos de nuevo en un vector que contenga la totalidad de elementos ya ordenados.

Ejemplo 6:

Utilizando el algoritmo de partición expuesto, dividir el siguiente vector de enteros en dos subvectores.

50 30 20 80 90 70 95 85 10 15 75 25

Inicialmente i=L=1, j=R=12, X=70 (uno de los valores centrales)
50 30 20 80 90 70 95 85 10 15 75 25 i=1, j=12, X=70

se mueve i mientras A(i)<70, y se mueve j mientras A(j)>70

```
            i=4                                    j=12
50 30 20 80 90 70 95 85 10 15 75 25 intercambio
50 30 20 25 90 70 95 85 10 15 75 80 i=5,j=11
```

como i<=j, vuelve a mover i mientras A(i)<70 y j mientras A(j)>70

```
               i=5                       j=10
50 30 20 25 90 70 95 85 10 15 75 80 intercambio
50 30 20 25 15 70 95 85 10 90 75 80 i=6,j=9
```

como aún i<=j, vuelve a mover i mientras A(i)<70 y j mientras A(j)>70

```
                  i=6          j=9
50 30 20 25 15 70 95 85 10 90 75 80 intercambio
50 30 20 25 15 10 95 85 70 90 75 80 i=7,j=8
```

como aún i<=j, luego vuelve a mover i mientras A(i)<70 y j mientras A(j)>70
y en esta ocasión el índice j pasa el 85, el 95 y se detiene en el 10 (i=6)

```
                  j=6 i=7
50 30 20 25 15 10 95 85 70 90 75 80
```

como no es cierto que sea i<=j entonces se acaban los bucles y hemos terminado la partición, con dos subvectores que ordenar: el de L a j y el de i a R

[50 30 20 25 15 10 **]** **[** 95 85 70 90 75 80 **]**

Nótese que el valor pivote no queda necesariamente en el centro ni en ningún sitio específico.

Como es fácil de prever, la generación sucesiva de subvectores, puede llegar a ser un problema, así si en un paso se elige el subvector izquierdo y se deja el derecho para más tarde, la particiones sucesivas nos complicarán la ordenación final. Un medio para evitar esos problemas es utilizar una estructura de datos tipo pila, en la que los datos se almacenan y recuperan por el mismo extremo.

El proceso comienza con un vector y se realiza la partición. Si el subvector derecho tiene más de un elemento, éste se coloca en la pila y se prosigue con la partición izquierda. Cuando se alcanza un punto en el que la partición izquierda tiene sólo un elemento, tomamos una partición de la pila y se continúa como antes. Cuando no existan particiones a sacar de la pila, la ordenación se ha terminado.

El algoritmo general será:

```
poner lista inicial en la pila
mientras haya particiones en la pila hacer
    tomar partición de la cima de la pila  {con forma de subvector: A(L) ... A(R)}
    mientras exista más de un elemento en la partición hacer
        partición
        si existe más de un elemento en partición derecha
          entonces poner partición derecha en la pila
        fin-si
        hacer partición izquierda de la lista
    fin-mientras
fin-mientras
```

Aunque la naturaleza de este algoritmo es recursiva y hacia este objetivo nos encaminamos, vamos a dar un algoritmo para su resolución iterativa utilizando una pila implementada mediante un vector (en realidad se trata de dos pilas conjuntas) que permite las siguientes acciones:

```
1.  {poner lista inicial en la pila}
        P ← 1  {P, puntero de la pila}
        SIzquierdo(P) ← 1
        SDerecho(P) ← N

2.  {poner partición derecha en la pila}
        P ← P + 1
        SIzquierdo(P) ← i
        SDerecho(P) ← R
```

3. {tomar una partición de la pila}
   L ← SIzquierdo(P)
   R ← SDerecho(P)
   P ← P - 1

4. {pila no vacía; hay particiones en la pila}
   P > 0

5. {hacer partición izquierda de la lista}
   R ← j

6. {hay más de un elemento en la partición}
   L < R

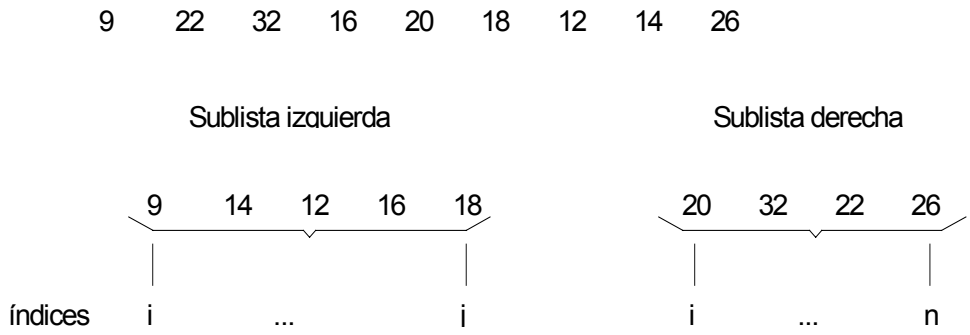
El algoritmo rápido incluyendo partición y ordenación es el siguiente:

```
algoritmo OrdenacionRapida
inicio
  P ← 1
  SIzquierdo(1) ← 1
  SDerecho(1) ← N
  mientras P > 0 hacer
    L ← SIzquierdo(P)
    R ← SDerecho(P)
    S ← S - 1
    mientras L < R hacer
      {partición de la lista}
      i ← L
      j ← R
      X ← A((L + R)/2)
      mientras i <= j hacer
          mientras A(i) < X hacer
              i ← i + 1
          fin-mientras
          mientras A(j) > X hacer
              j ← j - 1
          fin-mientras
          si i <= j
              entonces
                AUX ← A(i)
                A(i) ← A(j)
                A(j) ← AUX
```

```
            i ← i + 1
            j ← j - 1
          fin-si
      fin-mientras
      si i < R
         entonces
           P ← P + 1
           SIzquierdo(P) ← i
           SDerecho(P) ← R
      fin-si
      R ← j
    fin-mientras
  fin-mientras
fin
```

Como puede verse el método de ordenación rápida por métodos iterativos es enojoso y poco legible. La propia presentación que del algoritmo hemos hecho (dividir sucesivamente un vector en dos mitades, hasta reducirlas a un solo elemento) señala la utilización de la recursividad como método más apropiado. La recursión se puede aplicar, ya que cada vez se aplica el mismo método de partición o división a cada una de las subvectores. Se presentan dos problemas: 1) ¿Dónde se detienen las particiones sucesivas? y 2) ¿Cómo se elige el pivote para cada subvector? La primera respuesta, es cuando el subvector conste de un solo valor y para la segunda vamos a recurrir a la idea , ya vista, de utilizar un término central del vector como pivote.

Supongamos que contamos con un procedimiento OrdRápida (A, L, R), donde de acuerdo con la notación seguida hasta ahora, A es un vector y L y R son respectivamente el índice inferior y superior, del vector que se esta procesando en cada momento. Para convencernos de las posibilidades de la recursividad , lancemos este procedimiento con el siguiente vector (L = 1 y R = 9):



La ordenación de la parte izquierda implica el mismo procedimiento que en las particiones. Por consiguiente, procesar el subvector izquierdo supone el mismo procedimiento que antes, excepto que ahora: L = 1 y R = 5. La ordenación de la parte derecha implica el mismo procedimiento que antes, excepto que L = 6 y R = 9. Los procedimientos de ordenación son:

```
OrdRapida (A, L, J)
OrdRapida (A, I, R)
```

El algoritmo resultante, se lista a continuación:

```
procedimiento OrdRapida(A: vector[1..n] de real, L y R:enteros)
var
  i,j,Central,Aux: entero
inicio
  i ← L
  j ← R
  Central ← A[(L + R) div 2]     {división entera}
  repetir
    mientras A[i]  <  Central hacer
      i ← i +1
    fin-mientras
    mientras A[j]  >  Central hacer
      j ← j - 1
    fin-mientras
    si i <= j
      entonces
            Aux ← A[i]
            A[i] ← A[j]
            A[j] ← Aux
            i ← i + 1
            j ← j - 1
    fin-si
  hasta-que i > j
  {proceso de subvector izquierdo}
      si L < j entonces llamar_a OrdRapida (A, L, j)
  {proceso de subvector derecho}
      si i < R entonces llamar_a OrdRapida (A, i, R)
fin   {procedimiento OrdenarRapido}
```

La utilización de este procedimiento desde un programa para ordenar un vector V con n elementos se hará indicando que deseamos ordenar todo el vector, así

llamar_a OrdRapida(V, 1, n)

Aunque en el caso peor el algoritmo de ordenación rápida tiene orden de complejidad cuadrático, su nombre se debe a que en el caso medio su comportamiento es sensiblemente mejor que el de los métodos anteriores. Como ejemplo, dada una colección arbitraria de vectores de 100 elementos, el numero medio de comparaciones que utiliza el algoritmo de inserción es 2475, mientras que por ordenación rápida esta cantidad es de alrededor de 700.

7.3. EVALUACIÓN DE UN POLINOMIO

El cálculo del valor que toma un polinomio en un punto es una tarea muy frecuente, y que en muchas ocasiones debe hacerse de multitud de veces en un mismo programa. Por ello es importante disponer de un algoritmo eficiente.

7.3.1 EVALUACIÓN DIRECTA

Supondremos que los coeficientes del polinomio $P(x)=\sum_{i=0}^{n} a_i x^i$ los tenemos almacenados en el vector a(0),...a(n), y queremos calcular el valor del polinomio en el punto dado x. El primer algoritmo para resolver el problema se basa en su cálculo directo, calculando cada término en $a_i\ x^i$ y sumándolos

```
algoritmo polinomio1
   inicio
   P←a(0)
   desde j=1 hasta n hacer
     pot←x
     desde i = 2 hasta j
       pot←pot * x
     fin-desde
     P←P+a(j)*pot
   fin desde
fin
```

A pesar de su aparente simplicidad (es una implantación directa de la función polinómica) este algoritmo presenta un coste elevado, ya que la presencia de dos

bucles anidados, lo convierte en un algoritmo cuadrático, en función del numero de coeficientes. Además, al tratarse de un problema númerico, no podemos despreciar la generación de errores de redondeo que inevitablemente se producen, al efectuar un numero elevado de operaciones aritméticas.

7.3.2 ALGORITMO DE HORNER

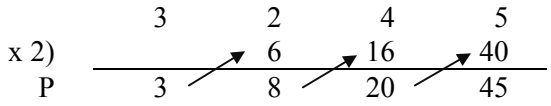
Afortunadamente, el calculo de la potencia en cada paso del bucle del algoritmo anterior se puede evitar, si recordamos que la escritura de un polinomio se puede hacer de la siguiente forma:

$$P(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n = (...(((a_nx + a_{n-1})x + a_{n-2})\ x + a_{n-3})... + a_1)x + a_0$$

Ejemplo 7:

Dado el polinomio: $P(x) = 3x^3 + 2x^2 + 4x + 5$ calcular su valor para x = 2, de la manera expuesta.

Esto significa hacer las siguientes operaciones:



De acuerdo con estos cálculos, P(3) = 45. Esta forma de evaluar el polinomio se conoce como método de Horner, y su algoritmo correspondiente se reduce a un solo bucle:

```
incio
P←a(n)
  desde j=n-1 hasta 0 hacer
  P←P*x+a(i)
  fin-desde
fin
```

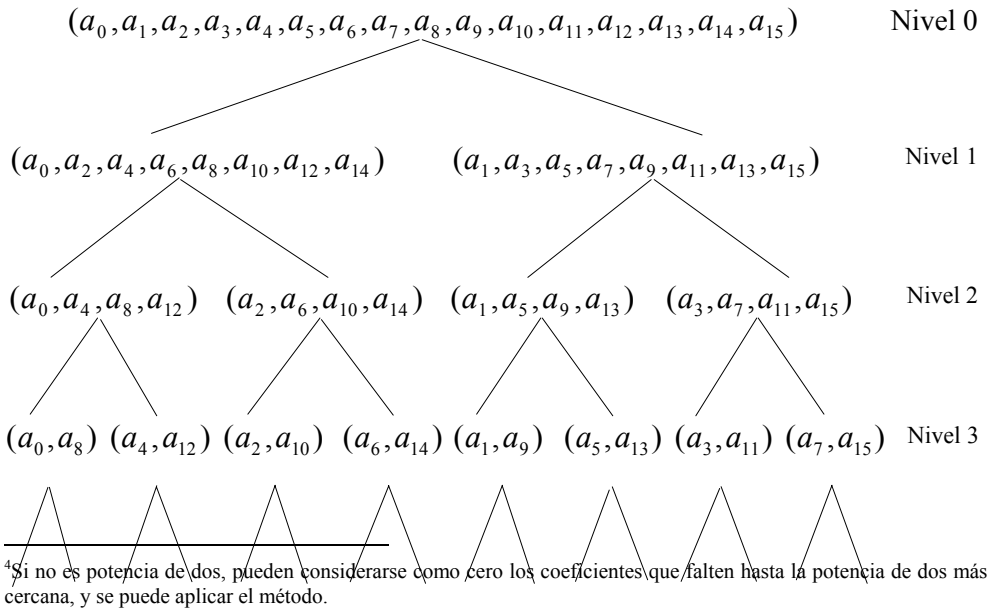
Este algoritmo es lineal ya que solo requiere *n* multiplicaciones, *n* sumas y *n+1* asignaciones.

7.3.3 MÉTODO DEL ÁRBOL

Aunque puede demostrarse que el método de Horner es óptimo con respecto al número de operaciones, al objeto de insistir en la variedad de métodos que existen para resolver un mismo problema, vamos a presentar otra forma de resolver el problema, basada en el "divide y vencerás" que si bien no es estrictamente mejor que el método de Horner para obtener el valor de un polinomio, puede superarlo en ciertos casos (p.e. cuando esta evaluación se tiene que hacer simultáneamente sobre un conjunto de puntos). Este método se basa en que todo polinomio se puede expresar separando los términos de potencia par e impar de la siguiente forma:

$$P(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n = (a_0 + a_2x^2 + a_4x^4 +...) + (a_1x + a_3x^3 + a_5x^5 +...)=$$
$$= Q(x^2)+xR(x^2) =Q(w)+xR(w), \qquad \text{donde } w= x^2$$

De esta manera, el problema de evaluar un polinomio de grado *n*, se ha transformado en evaluar dos polinomios de grado *n/2*, los polinomios Q y R. Si suponemos, sin pérdida de generalidad, que el polinomio inicial tiene un número de coeficientes que es potencia de dos[4] ($n=2^k-1$), el procedimiento anterior lo podemos aplicar recursivamente *k* veces, hasta conseguir que los polinomios generados en el último paso, sean de grado cero (o sea, queden reducidos a los coeficientes del polinomio). Representando los sucesivos polinomios sólo por sus coeficientes, este proceso lo podemos representar por un árbol binario, como se muestra en el siguiente diagrama:



[4]Si no es potencia de dos, pueden considerarse como cero los coeficientes que falten hasta la potencia de dos más cercana, y se puede aplicar el método.

a_0 a_8 a_4 a_{12} a_2 a_{10} a_6 a_{14} a_1 a_9 a_5 a_{13} a_3 a_{11} a_7 a_{15} Nivel 4

Para evaluar el valor de un polinomio en un punto, basta con que evaluemos cada nodo del árbol, desde las hojas a la raíz (*bottom-up*), de la siguiente forma:

Valor(nodo)=Valor(hijo_izquierdo(nodo))+Z*Valor(hijo_derecho(nodo)),
donde $Z=X^{2^{Nivel(nodo)}}$

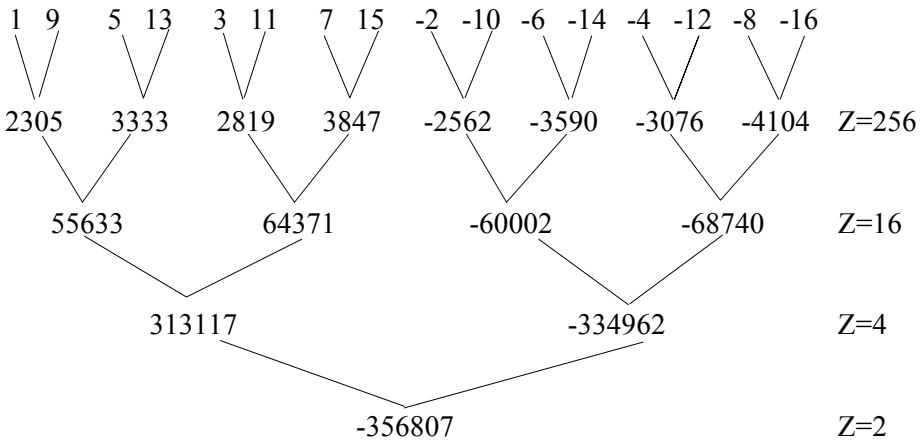
Obsérvese que en el ejemplo dado se han de evaluar 15 nodos, y cada evaluación implica una asignación, una suma y una multiplicación. Por tanto este método es comparable en complejidad computacional al de Horner, salvo en dos cosas: La primera es que se necesitan tres multiplicaciones para generar x^2, x^4 y x^8 y la segunda y más importante, es que hay que disponer los coeficientes del polinomio en el orden presentado en el Nivel 4.

Vamos a ilustrar con un ejemplo concreto la ventaja que puede suponer este método respecto al de Horner

Ejemplo 8:

Calcular el valor que toma en el punto x=2 el polinomio utilizando el método del árbol

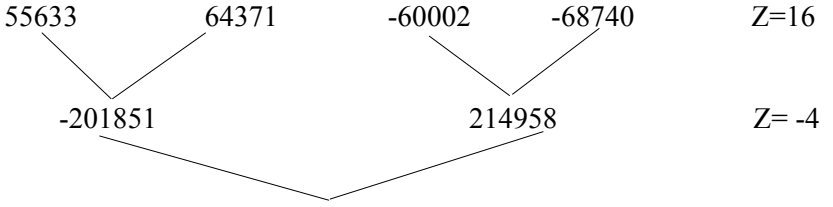
$P(X) = 1-2x+3x^2-4x^3+5x^4-6x^5+7x^6-8x^7+9x^8-10x^9+11x^{10}-12x^{11}+13x^{12}-14x^{13}-15x^{14}-16x^{15}$



Si quisiéramos calcular el valor del polinomio en el punto x= -2, sólo tenemos que hacer:

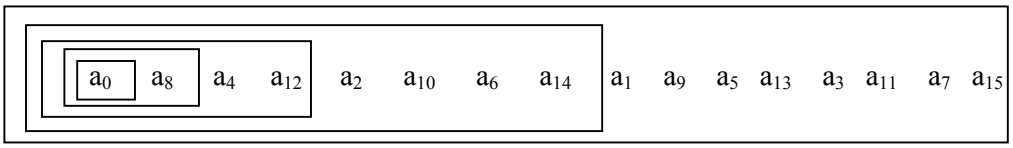
P(-2)= 313117+2*334962= 983041

dado que el cuadrado de 2 y -2 coinciden. Si quisiéramos calcular el valor del polinomio también en los puntos $2i$ y $-2i$, sólo necesitamos hacer 4 multiplicaciones y 4 sumas:



Sin embargo, para calcular P(-2), P(2i) y P(-2i) por el método de Horner, tendríamos que haber empezado desde el principio. Por consiguiente, este método puede ser más conveniente que el de Horner, si necesitamos evaluar el polinomio en varios puntos y coinciden sus cuadrados, ó potencias cuartas, etc...

Queda pendiente el problema de distribuir los coeficientes según el orden que figura en las hojas del árbol, pero esto es sencillo si observamos lo que ocurre en el ejemplo visto, del polinomio de grado 16, donde el subíndice de los coeficientes que en la siguiente figura están entre una caja y otra, es igual -en el mismo orden- que el de los subíndices de las cajas inferiores más dos elevado a la diferencia entre N y el número de cajas exteriores (siendo N el grado del polinomio más uno):



Un algoritmo para generar un vector que contenga los subíndices en el orden adecuado, es pues el siguiente:

P(0)←0
C←N
K←1
mientras k<>N **hacer**

```
        C←C/2
        desde j=k hasta 2*k-1 hacer
            P(j) ←P(j-k)+C
        fin-desde
    k←k*2
    fin-mientras
```

Como podemos observar la variedad de algoritmos, para resolver un tema tan aparentemente sencillo como éste, es realmente llamativa y esta es una circunstancia que no puede ni debe ser ignorada durante el proceso de programación.

7.4. ALGORITMOS PARA LA BÚSQUEDA DE CADENAS DE CARACTERES

Hemos abordado la búsqueda de una determinada información en un entorno ordenado, estudiando diversos algoritmos y observando como todos ellos sacaban provecho de este orden ya establecido para hacer una búsqueda eficiente. Sin embargo, en ocasiones es necesario buscar en un entorno en el que no se ha producido previamente una ordenación, y en el que además no se busca una coincidencia exacta, sino que se busca que en qué posición aparece un determinado patrón (o bien si aparece o no). Ejemplos de tales situaciones se producen en un procesador de textos cuando buscamos una palabra o parte de ella, y en algunos sistema operativos, que permiten buscar en qué ficheros aparece una determinada secuencia de caracteres, sin olvidar su uso en programación cuando trabajamos con variables de cadena. Esta es una tarea tan importante que en casi todos los casos anteriores se suelen suministrar órdenes o instrucciones que la realizan. Vamos a estudiar algoritmos para buscar una secuencia de caracteres en un cadena, obteniendo el lugar a partir del cual aparece esta subcadena o bien un cero si no existe tal secuencia.

7.4.1 ALGORITMO DE COMPARACIÓN

El algoritmo más directo, y que de hecho se empleó durante tiempo hasta obtener uno mejor, consiste en comparar el primer carácter de la cadena con el primero de la secuencia; si coinciden se pasa analizar el segundo, si coinciden el tercero y así sucesivamente hasta que o coincide el último de la secuencia (momento en el cual acaba la búsqueda) o se encuentra un carácter distinto, momento en el cual se pasa a comparar el segundo carácter de la cadena con el primero de la secuencia; si hay coincidencia se comparan el tercero con el segundo, y así sucesivamente.

Ejemplo 9:

Veamos un ejemplo de este algoritmo para localizar “trigales” en un texto que comienza por “tres tristes tigres comieron trigo en unos trigales...”:

```
TRES_TRISTES_TIGRES_COMIERON_TRIGO_EN_UNOS_TRIGALES ...
TRIGALES              TRIGALES
 TRIGALES              TRIGALES
  TRIGALES              TRIGALES
   TRIGALES              TRIGALES
    TRIGALES              TRIGALES
     TRIGALES              TRIGALES
      TRIGALES              TRIGALES
       TRIGALES              TRIGALES
        TRIGALES              TRIGALES
         TRIGALES              TRIGALES
          TRIGALES              TRIGALES
           TRIGALES              TRIGALES
            TRIGALES              TRIGALES
             TRIGALES              TRIGALES
              TRIGALES              TRIGALES
               TRIGALES              TRIGALES
                TRIGALES              TRIGALES
                 TRIGALES              TRIGALES
                  TRIGALES              TRIGALES
                   TRIGALES              TRIGALES
                    TRIGALES              TRIGALES
                     TRIGALES              TRIGALES
```

En negrita se han puesto las comparaciones con resultado exitoso, y subrayadas las que resultan fallidas. Como puede observarse, se han de hacer 53 comparaciones para llegar a encontrar la solución y 8 más para comprobar que es ésta. En el peor caso la complejidad del algoritmo es del orden del producto de las longitudes de las cadenas, no obstante en la práctica este peor caso no se suele presentar, dado que este implica buscar en una cadena del tipo “aaaaaaaaaaa...b” un patrón del tipo “aaaaa...b”, lo que afortunadamente no es muy habitual. Esta complejidad lineal con el tamaño de la cadena larga, resulta absolutamente ineficiente cuando se ha de emplear para buscar entre cantidades de información cercanas o superiores al orden de Mbytes. Por ello resulta fundamental obtener algoritmos más eficientes.

El algoritmo correspondiente, lo podemos representar como sigue:

función posición(cadena_corta, cadena_larga):número
N←**longitud**(cadena_corta)

```
M←longitud(cadena_larga)
j←1
k←1
mientras (j ≤ N) y (k ≤ M) hacer
      si cadena_corta(j)=cadena_larga(k)
            entonces
            j←j+1
            k←k+1
            si_no
            k←k-j+2
            j←1
      fin-si
fin_mientras
si j=N+1
      entonces
      posición←k-j+1
      si_no
      posición←0
fin-si
fin-funcion.
```

7.4.2 ALGORITMO DE BOYER-MOORE

Partamos de la idea de ir avanzando sobre la cadena larga, pero haciendo las comparaciones con la secuencia en sentido inverso, esto es, desde su último carácter al primero. Así, si la longitud de la secuencia es N, se compara el carácter N-ésimo de la cadena larga, con el último de la secuencia, si coinciden se pasa a analizar el anterior y así sucesivamente hasta que coincida la subcadena completa o hasta que encontremos un carácter distinto. En este caso la secuencia no se "desliza" un lugar sobre la cadena, sino que se desliza hasta el carácter de la secuencia que coincide con el carácter distinto, antes encontrado. Caso de no existir este carácter en la secuencia, ésta se deslizaría totalmente empezando el nuevo proceso a partir de esta posición.

Ejemplo 10:

Aplicar el algoritmo de Boyer-Moore para el caso en que la cadena cadena_larga="campanarios" (M=11) y la secuencia, cadena_corta="rios" (N = 4).

En primer lugar se compararía la cuarta letra de la secuencia larga, “p”, con la última letra de la secuencia, “s”. Al ser distintas, y además la “p” no aparecer en la secuencia, se intenta buscar la secuencia a continuación de la “p”. Para ello consideraríamos la cuarta letra, a partir de la “p”, esto es la “r”, para compararla de nuevo con la “s”, resultando también ser distintas. Sin embargo, ahora si la “r” aparece en la secuencia, por lo que se pondría las dos “r” alineadas y a partir de ellas, las comparaciones, letra a letra, resultarán ciertas. Veamos este algoritmo aplicado al ejemplo anterior (M = 51 y N = 8):

```
TRES_TRISTES_TIGRES_COMIERON_TRIGO_EN_UNOS_TRIGALES...
TRIGALES (I ⇒ hace avanzar 5)
     TRIGALES (_ ⇒ 8)
             TRIGALES (C ⇒ 8)
                     TRIGALES (_ ⇒ 8)
                             TRIGALES(N ⇒ 8)
                                     TRIGALES (R ⇒ 6)
                                             TRIGALES
```

Como puede observarse en este caso sólo han sido necesarias 6 comparaciones para llegar a encontrar la solución y 8 más, para comprobar que es ésta, por lo que la mejora de este algoritmo es más que notable. Con este método, si la cadena tiene M caracteres y la secuencia N, en la práctica se han de hacer alrededor de M/N comparaciones hasta encontrar la solución.

La formulación del algoritmo correspondiente exige que podamos definir la cantidad de caracteres que se tiene que deslizar la secuencia para cada posible carácter de la cadena. Esto es fácil de hacer utilizando un vector cuyos índices se correspondan con dichos caracteres, como se muestra a continuación:

```
Algoritmo avanza(texto:cadena; var incr:vector);
inicio
    N←longitud(texto)
    desde letra='_' hasta 'z' hacer
        incr[letra]←N
    fin-desde
    desde i=1 hasta N hacer
        incr[texto[i]]←N-i
    fin-desde
fin
```

Por motivos de simplicidad, en el algoritmo anterior estamos usando la posibilidad de utilizar los caracteres como índices, cosa que no todos los lenguajes de programación soportan, sin embargo siempre es posible convertir un carácter a un numero y utilizarlo como índice (Recuérdese que en ASCII, los caracteres especiales y alfanuméricos van desde el carácter blanco, de valor decimal 32, hasta

z de valor decimal 122). Así el vector *incr* es del tipo array[alfabeto] de enteros, siendo alfabeto = '_'.....'z'.

El algoritmo final de búsqueda de secuencias de Boyer- Moore es:

```
función busca(texto1,texto2:cadena):entero
inicio
M ← longitud(texto2)
N ← longitud(texto1)
i ← N
j ← N
avanza(texto1,incrementa);
mientras  (i<=M) y (j>0) hacer
      si texto1[j]=texto2[i]
            entonces
            i ← i-1
            j ← j-1;
            si_no
            j ← N
            i ← i+incrementa[texto2[i]];
      fin-si
fin_mientras
si j=0
      entonces
      busca ← i+1
      si_no
      busca ← 0
fin-si
fin-función.
```

7.5. NOTAS FINALES SOBRE COMPLEJIDAD

7.5.1 ALGORITMOS NO-DETERMINISTAS

El concepto de algoritmo que hemos manejado hasta el momento, implica que el resultado de cualquiera de sus operaciones está definido de forma única, y por tanto distintas ejecuciones del mismo producen el mismo resultado. Estos algoritmos están perfectamente adaptados para su codificación en un lenguaje de alto nivel y su ejecución en un computador. Sin embargo, podemos extender el concepto de algoritmo para permitir que el resultado de un paso no esté determinado de forma única, sino que tenga un conjunto calculable de posibles

elecciones. Estos algoritmos se conocen como no-deterministas, y los utilizamos frecuentemente:

- Cuando vamos de un punto a otro de la ciudad, no siempre seguimos el mismo camino, y esto es porque las distintas calles a seguir las elegimos de forma no determinista.

- El método que utilizamos cada uno para jugar al ajedrez. Ante la imposibilidad de analizar todas las posibles jugadas y quedarnos con la más ventajosa (lo que daría un algoritmo determinista), nosotros elegimos algunas de las posibles para analizar, de forma no determinista (de hecho ante una misma disposición de piezas, podríamos dar respuestas distintas en distintas partidas).

Por tanto, un algoritmo no-determinista debe tener al menos un paso en el que se genere una posible hipótesis de solución del problema, y debe contener algún método que le permita comprobar si esta hipotética solución se confirma como tal solución del problema (momento en el cual acaba con éxito el algoritmo) o si por el contrario no lo es (acabando con un fallo, tras lo cual se podría volver a intentar aplicar el algoritmo). Por ello, los algoritmos no-deterministas se consideran basados en el paradigma de programación conocido como "generación y prueba" (*generate and test*)[5] y su complejidad tiene que ser analizada de diferente forma a como lo hemos hecho en el caso determinista, ya que si intentamos aplicar la misma definición nos encontramos con el problema de que el concepto de número de operaciones elementales requeridas para alcanzar una solución no es directamente trasladable a los algoritmos no-deterministas, puesto que no siempre realizan el mismo numero de pasos con los mismos datos iniciales. La definición de la función f(n) para estos algoritmos es la siguiente: dado un problema de tamaño n, f(n) es el mínimo número de operaciones elementales requerido para alcanzar la solución del problema, si existe la sucesión de elección de hipótesis que llevan a ella.

7.5.2 PROBLEMAS DE CLASE P Y NP

Para muchos de los problemas prácticos, desgraciadamente, no se conoce un algoritmo polinomial que los resuelva, es más hay muchos de ellos en los que el mejor algoritmo (determinista) conocido, tiene una complejidad exponencial. Sin embargo es posible encontrar un algoritmo no-determinista que tenga complejidad polinomial, y que en muchos casos lo hace resoluble a efectos prácticos. Ello ha llevado a definir dos clases de problemas: los de la **clase "P"** (para los que existe un algoritmo determinista de complejidad Polinomial) y los de la **clase "NP"** (para

[5] Pensemos sólo en algoritmos de decisión no-deterministas, que son aquellos que pueden dar una respuesta sí o no (exito o fallo) a un determinado problema.

los que existe un algoritmo No-determinista de complejidad Polinomial)[6], o abreviadamente también se habla de problemas P y problemas NP. Dado que si un problema está en P, existe un algoritmo determinista de complejidad polinomial que lo resuelve, a partir de este algoritmo se puede generar uno no-determinista que también tenga complejidad polinomial, simplemente tomando como método de generación de hipótesis el resultado de la aplicación del algoritmo determinista. Por tanto podemos afirmar que la clase P está incluida en la clase NP.

A la luz de la tabla 7.1, queda claro que es de gran importancia para establecer la computabilidad de un problema distinguir si está en P o es NP pero no está en P (Dado que en el primer caso se puede resolver en un tiempo "razonable" y en el segundo resultaría intratable). Sin embargo, hasta el momento nadie ha sido capaz de demostrar que las clases P y NP sean distintas (o sea, nadie ha podido encontrar un problema que esté en NP-P, o bien demostrar que esto es imposible), siendo uno de los grandes problemas que quedan abiertos.

Como conclusión, queda la idea de que con el computador no todos los problemas son abordables y por tanto, cada vez que nos planteemos un algoritmo, sea de la clase que sea, lo primero a considerar son sus posibilidades reales de ser resuelto a través de un programa de computador, ya que por mucho que mejore la tecnología en el futuro, la naturaleza de la solución propuesta la puede hacer no computable.

7.5.3 INTRODUCCIÓN A LAS MAQUINAS DE TURING Y A LOS PROBLEMAS ALGORITMICAMENTE IRRESOLUBLES

Hasta este momento hemos tratado problemas para los que existe un algoritmo (sea determinista o no) que los resuelve, aunque como hemos visto esa solución sea en muchos casos inviable en la práctica por el desmesurado tiempo de computación requerido. Sin embargo, no hemos abordado hasta ahora si para cualquier problema podemos encontrar un algoritmo que lo resuelva, o si por lo contrario, hay problemas que no son resolubles algorítmicamente. Esta motivación llevó a Alan Turing en 1936 (como puede observarse antes de que existiesen los computadores en su concepción actual, y antes de que existiese el concepto de lenguaje de programación) a definir una herramienta teórica, conocida como **máquina de Turing**, que le permitiera analizar las limitaciones de los algoritmos.

Una máquina de Turing está compuesta por una unidad de control de estados finitos y una cinta, que se supone dividida en tramos, e infinita en ambas direcciones. Cada tramo contiene un símbolo perteneciente a un alfabeto finito prefijado. La comunicación entre la unidad de control y la cinta está permitida sólo a través de una cabeza de lectura/escritura que es capaz de leer símbolos de la cinta

[6] Por extensión, se habla de algoritmos P y algoritmos NP.

y escribirlos, así como de moverse en ambas direcciones a un tramo de la cinta adyacente al actual. La unidad de control opera en pasos discretos, y en cada paso sus acciones dependen del estado en el que esté y del último símbolo leído de la cinta (ver Figura 7.4).

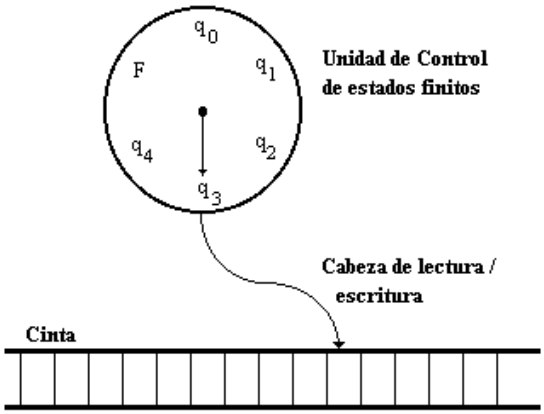


Figura 7.4 *Representación de una Máquina de Turing*

Para definir una máquina de Turing, además de los símbolos del alfabeto y los posibles estados, es necesario definir, para cualquier combinación de estado actual y cualquier símbolo leído, tres valores:

1. El estado siguiente que corresponde a esa combinación.
2. El símbolo a escribir en la cinta, correspondiente a esa combinación.
3. El movimiento de la cinta que corresponde a esa combinación. Este movimiento sólo puede tomar tres valores: quedarse en el tramo actual, moverse al tramo adyacente derecho y moverse al tramo adyacente izquierdo.

La máquina comienza a funcionar cuando se pone en un determinado estado que llamaremos "arranque" y se para al alcanzar un estado que llamaremos "parada" (no tiene por qué ser único). El funcionamiento de la máquina se rige por el siguiente ciclo:

Paso 1: La máquina lee un símbolo de la cinta, si no está en un estado de parada.
Paso 2: Dado el estado actual y el símbolo leído, escribe en la cinta el símbolo correspondiente a esa combinación.
Paso 3: Dado el estado actual y el símbolo leído, pasa al nuevo estado correspondiente a esa combinación.

Paso 4: Dado el estado actual y el símbolo leído, mueve la cabeza de lectura según el movimiento correspondiente a esa combinación, y vuelve al paso 1.

Como puede observarse hay una gran similaridad en la arquitectura y en la forma de trabajar entre un computador y una máquina de Turing, si bien este dispositivo teórico supone una simplificación respecto a un computador. Así, la unidad de control de estados finitos desempeña el papel de la CPU, y la cinta el papel de la memoria RAM en la que se guardan los datos (se supone que inicialmente hay un conjunto finito de símbolos puestos sobre la cinta, estando el resto de la cinta "en blanco"). Por otra parte, un programa para una máquina de Turing, viene determinado por los tres valores correspondientes a cada combinación estado actual / símbolo leído, que como podrá observarse en el siguiente ejemplo permite representar algoritmos (evidentemente con una adecuada elección de estados y del alfabeto a utilizar).

Ejemplo 11:

Definir una máquina de Turing que detecte si un número binario tiene ceros sobrantes al principio del número, y que en este caso los borre. Se supone que cada dígito está en un tramo de la cinta, de forma consecutiva, y que el cabezal de lectura está situado en el espacio anterior al último dígito (el menos significativo del número), posición en la que el programa debe dejarlo al finalizar.

En este problema el alfabeto será {0,1,b}, donde 0 representa el número cero, 1 el número 1 y b blanco en la cinta. Los estados a definir pueden ser estos ocho, que se pueden interpretar de la forma siguiente:

Ar: estado en el que se encuentra el programa al inicio
q_o: "ir a la izquierda hasta encontrar el primer dígito del número"
q_1: "mirar primer dígito"
q_2: "seguir borrando"
q_3: "ir a la posición inicial, sabiendo que no se han borrado dígitos"
q_4: "ir a la posición inicial, sabiendo que se han borrado dígitos"
Si: "Si que sobran dígitos"
No: "No sobran dígitos"

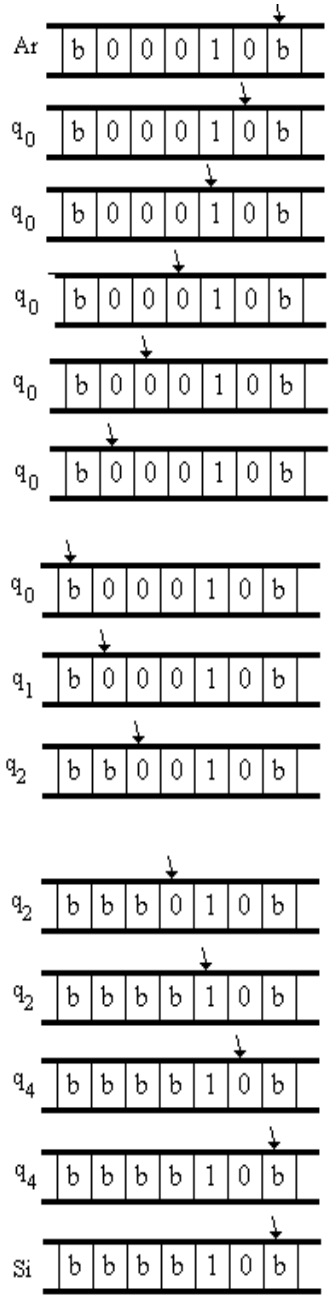
El estado inicial o de arranque es el "Ar" y los estados "Si" y "No" se consideran como estados de parada, cada uno de ellos dando la respuesta correspondiente. El "programa" que completa la definición de la máquina de Turing para este ejemplo, viene dado por la siguiente tabla:

Estado Actual	Símbolo leído	Valor a escribir	Nuevo estado	Movimiento
Ar	b	b	q_o	Izquierda
q_o	0	0	q_o	Izquierda
q_o	1	1	q_o	Izquierda
q_o	b	b	q_1	Derecha
q_1	0	b	q_2	Derecha
q_1	1	1	q_3	Derecha
q_1	b	b	No	No_mover
q_2	0	b	q_2	Derecha
q_2	1	1	q_4	Derecha
q_2	b	b	Si	No_mover
q_3	0	0	q_3	Derecha
q_3	1	1	q_3	Derecha
q_3	b	b	No	No_mover
q_4	0	0	q_4	Derecha
q_4	1	1	q_4	Derecha
q_4	b	b	Si	No_mover

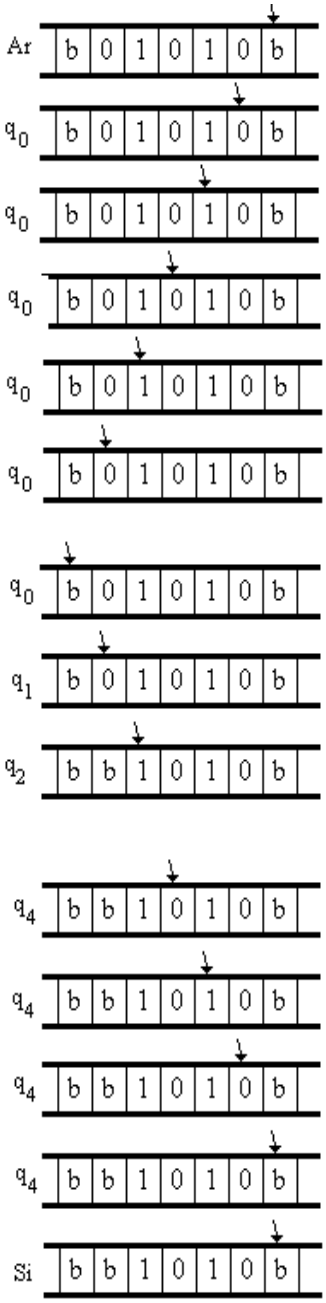
En la página siguiente podemos seguir el resultado de aplicar esta máquina de Turing a distintos datos de entrada.

Como puede observarse con la evolución del cabezal, de los estados y del contenido de la cinta, el diseño de un programa con una máquina de Turing se asemeja mucho al de un algoritmo, si bien en la máquina de Turing disponemos de instrucciones mucho más elementales de las que estamos acostumbrados a manejar, lo cual hace más complejo el trabajo de construcción del algoritmo. Sin embargo, con una máquina de Turing es posible realizar tareas mucho más complicadas que las del ejemplo expuesto, y existen estudios de cómo una máquina de Turing puede realizar los trabajos que hace un computador en su concepción actual.

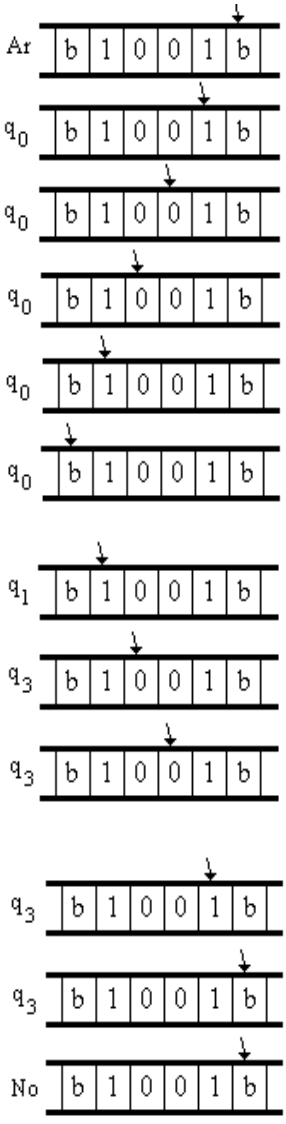
Caso 1: número 00010



Caso 2: número 01010



Caso 3: número 1001



El gran resultado de Turing se basa en la conjetura (conocida como Tesis de Church) de que la potencia computacional de la máquina de Turing es al menos como la de los algoritmos (y por tanto todo problema que no pueda resolverse con la máquina de Turing tampoco tendrá solución algorítmica). Más concretamente, Turing demostró que existen problemas que no tienen solución con su máquina. Su resultado se basa en el problema conocido como "problema de parada de una máquina de Turing", cuyo enunciado es el siguiente:

> *Problema de Parada de la Máquina de Turing:*
> *Dada una máquina de Turing M y una cinta C, ¿puede definirse una máquina de Turing que decida si la máquina M parará al procesar la cinta C?*

Al demostrar que ello no es posible, Turing sentó las bases para demostrar que otros problemas tampoco tienen solución algorítmica[7] (o empleando otras terminologías también se dice que son **irresolubles** o que no son decidibles, dado que no se puede contestar ni que sí, ni que no). La técnica empleada se basa en reducir estos problemas al problema de parada de una máquina de Turing, y ha sido utilizada para demostrar la irresolubilidad de otros problemas como sería el caso de cuestiones sobre gramáticas en teoría de lenguajes, de teselamiento del plano en geometria (dada una figura, saber si es posible construir un mosaico sobre una superficie dada), de decidibilidad del cálculo de predicados de primer orden en lógica, etc.

[7] Este resultado, con lo expuesto hasta este momento, sólo es extensible a algoritmos deterministas; sin embargo, es fácil definir una máquina de Turing *no-determinista* que capture el concepto de algoritmo no-determinista, y se puede demostrar que estas máquinas son equivalentes, al poderse transformar una en la otra, con un algoritmo de complejidad polinomial. Por tanto, los resultados sobre irresolubilidad son también extensibles a los algoritmos no-deterministas.