9 Análisis de la Complejidad de los Algoritmos

Con el aumento creciente de la velocidad de los procesadores, de las memorias y de otros dispositivos electrónicos, se pueden solucionar ahora problemas para los que antes los programas de solución resultaban lentos o para los que ni siquiera se podía intentar su solución. Pero esto no basta, hay problemas para los que las velocidades de las computadoras actuales, y las de la que están por venir, no son suficientes. Por otra parte surgen nuevas necesidades de aplicaciones que requieren de respuesta rápida. La psiquis del actual modelo de usuario interactivo requiere de respuesta inmediata a situaciones que antes o no se enfrentaban, o para las que había que tener la paciencia de esperar por la respuesta.

En los capítulos anteriores se ha visto, y se seguirá estudiando en los capítulos que siguen, como el método orientado a objetos nos ayuda a modelar y diseñar mejor. La orientación a objetos nos facilita y hace más confiable la implementación de los programas, así como también nos ayuda a reutilizar, adaptar y modificar soluciones ya existentes. Todo esto redunda en mas confiabilidad y en una disminución del costo de producción de los productos de software. Ciertamente éste era un aspecto muchas veces soslayado cuando el hombre "trabajaba para la computadora" pero no ahora en que somos más conscientes y pudientes de que queremos a la computadora "trabajando para el hombre".

Todo esto es solo una parte importante de la historia. La otra cara de la moneda es que cuando se ejecuta un programa se espera que el programa termine en un tiempo razonable según nuestras necesidades y requerimientos. En lo fundamental este tiempo es independiente del lenguaje de programación y del método de programación (orientado a objetos o no).

Como ya se ha dicho, un *algoritmo* es un conjunto "claramente" especificado de instrucciones (lo más claramente en nuestro caso es el algoritmo expresado en C#) encaminadas a la solución de un problema. Una vez que se ha concebido o dispone de un algoritmo y se ha determinado que es correcto (y en esto el método orientado a objetos y un buen lenguaje de programación que lo soporte juegan un importante papel), el siguiente paso es determinar el monto de recursos, tales como espacio de memoria y tiempo de ejecución, que el algoritmo requiere. Esto se denomina *complejidad del algoritmo*.

No es objetivo de este libro suplantar la voluminosa literatura que existe sobre este tema (a la cual el lector interesado puede dirigirse). La intención en este capítulo es dar una introducción general a este importante asunto, sobre todo en esta época en que algunos erróneamente piensan que la programación se reduce a la confección de amistosas y atractivas interfaces de usuario. En este capítulo se estudiará cómo estimar el tiempo de ejecución requerido por un algoritmo y se verá cómo se puede reducir significativamente este tiempo de ejecución para algunos algoritmos.

Draft para estudiantes de MATCOM UH

9.1 ¿Qué es la Complejidad de un Algoritmo?

Por lo general el tiempo que demora un algoritmo en ejecutar depende del volumen de datos que debe procesar. Se supone que si se quiere ordenar alfabéticamente un listado de 10,000 nombres esto requiera más tiempo que si se quiere ordenar uno de 100 nombres. Se puede decir que el tiempo de ejecución de un algoritmo es una función que depende de la cantidad de datos sobre las que opera el algoritmo (lo que se conoce como *tamaño de la entrada*). El lector, a veces acostumbrado a la mentalidad interactiva, no debe confundirse: el tamaño de la entrada no quiere decir necesariamente "teclear" un gran volumen de datos, los datos pueden provenir de otra fuente como un archivo, una conexión en la red o pueden ser generados por la propia aplicación. El valor exacto de esta función, que da el tiempo que se demora la ejecución del algoritmo, depende de factores tales como la velocidad del procesador y la calidad del código que genera el compilador. Estos son factores que están determinados por el compilador y el procesador que se utilicen. El lector puede medir, utilizando por ejemplo la clase `Cronómetro` que se ha estudiado en este libro, el tiempo real que demora la ejecución en su computadora de los programas que ha desarrollado.

Sin embargo, lo que se pretende con el análisis de la complejidad de los algoritmos es determinar cómo estimar el costo de ejecución en tiempo y memoria del algoritmo seleccionado con respecto al volumen de datos a procesar por éste (con independencia del compilador y de la velocidad del procesador a utilizar) y a la vez estudiar cómo lograr posibles mejoras a los algoritmos.

Tome como ejemplo el problema de "descargar" un fichero de Internet. Considere que hay un costo inicial de 5 segundos por establecer la conexión y que después la descarga prosigue a una velocidad de 1.5 Kilobytes por segundo. Entonces, si el archivo tiene n Kilobytes, el tiempo de descargar todo el fichero es: $T(n) = n/1.5 + 5$ segundos. Podemos calcular entonces que mientras el costo de descargar un archivo de 100 Kb es de 71.6 (100/1.5+5) segundos, el de descargar uno de 200 K es de 138.3 (200/1.5+5) segundos. Es decir, hablando groso modo, si duplicamos el tamaño de la entrada duplicamos el costo de ejecución. Note que si se triplicase la velocidad de transmisión (con un mayor ancho de banda por ejemplo) el costo real de descargar los archivos anteriores (si descontamos el tiempo inicial de conexión que no depende de la velocidad de transmisión ni de la cantidad de datos) se divide por tres, es decir serían 27.2 y 49.4 segundos. Esto es análogo a lo que ocurre cuando se buscan procesadores más rápidos, se disminuye el costo real claro está, pero lo que no varía, como en el caso de este ejemplo, es que *si se duplica el volumen de la entrada se duplica el tiempo de ejecución*. Es decir, la *relación* entre el volumen de la entrada y el tiempo de ejecución no depende de la velocidad del procesador. Es esto lo que caracteriza de manera intrínseca el costo de ejecución del algoritmo.

Lo que se pretende con el análisis de la complejidad de los algoritmos es determinar este *costo* del algoritmo a la vez que tratar de encontrar algoritmos alternativos que resuelvan el mismo problema con menor costo.

Un algoritmo, como el del ejemplo anterior, en el que el tiempo de ejecución es directamente proporcional al volumen de la entrada se denomina un *algoritmo lineal*. En este tipo de algoritmos el costo se representa con una expresión de la forma $nc_1 + c_2$. En

Estudiantes de MATCOM UH

el ejemplo de arriba c_2 es el tiempo inicial de preparación de la conexión, c_1 es la velocidad de transmisión y n es el volumen de los datos. Sin embargo, aunque puede ser útil disminuir los valores de las constantes c_1 y c_2 (un procesador más rápido, una mayor velocidad de transmisión, un menor tiempo para establecer conexión, un compilador que genere un mejor código) lo que se quiere destacar aquí es que el tiempo de descargar el fichero es de todos modos proporcional a su tamaño. En la literatura de análisis de la complejidad de los algoritmos se utiliza una notación conocida como *notación Big-O* para representar este factor de crecimiento del costo en función del volumen de los datos lo que se denomina *orden* del algoritmo. De manera abreviada se dice que en el caso de este ejemplo el *algoritmo es lineal* o de *orden n* y se denota por $O(n)$.

Formalmente decimos que la función $f(n)$ es $O(g(n))$ y se escribe $f(n) = O(g(n))$ o $f(n) \in O(g(n))$, si existen n_0 y c tal que para todo $n \geq n_0$ se cumple que $f(n) \leq cg(n)$ como se ilustra en la Figura 9-1.

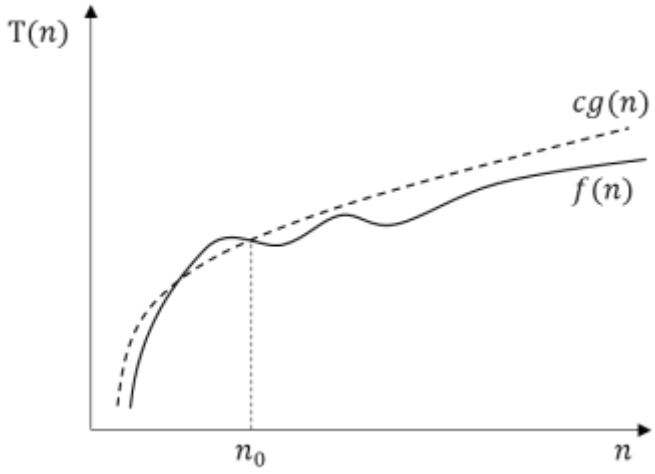


Figura 9-1. Análisis de la cota superior asintótica

En el caso anterior se tiene la función de costo $T(n) = nc_1 + c_2 \leq n(c_1 + 1)$ para todo $n \geq n_0 = c_2$. Podemos determinar entonces que nuestro costo es $O(n)$.

El caso de un algoritmo lineal es, en principio, el escenario ideal. Nuestro "sentimiento de justicia" acepta intuitivamente que si el volumen de los datos a procesar aumenta, entonces el tiempo de ejecución aumente de modo linealmente proporcional. Sin embargo, esto no es siempre así para todos los problemas y algoritmos. En el resto de este capítulo se estudiarán algoritmos cuyo orden es ¡menor que lineal!, por ejemplo $O(\log_2 n)$; pero también se verán algoritmos cuyo orden es mayor, por ejemplo $O(n^2)$ y $O(n^3)$, es decir que el tiempo es cuadráticamente o cúbicamente proporcional al volumen de la entrada. Lo "trágico", es que también existen problemas y algoritmos de solución que son de orden $O(2^n)$ u $O(n!)$. En tales escenarios un simple incremento en el volumen de los datos participantes en el procesamiento puede hacer que un algoritmo sea impracticable de ejecutar aún en las computadoras actuales (y futuras) más veloces. La Tabla 9-1 nos muestra una tabla en orden creciente de las funciones de costo más comunes y un aproximado del tamaño de la entrada que pudiera ser resuelta en un procesador hipotético de un millón de instrucciones por segundo antes de determinado tiempo.

Draft para estudiantes de

Familia de funciones	Nombre	1 min	1 día	1 año	1000 años
$O(1)$	Constante	∞	∞	∞	∞
$O(\log n)$	Logarítmica	$2^{60000000}$	$2^{86000000000}$	$2^{31000000000000}$	$2^{3\cdot 10^{16}}$
$O(n)$	Lineal	$6 \cdot 10^7$	$8.6 \cdot 10^{10}$	$3.1 \cdot 10^{13}$	$3.1 \cdot 10^{16}$
$O(n \log n)$	n logaritmo de n	$4 \cdot 10^6$	$3.9 \cdot 10^9$	$1.1 \cdot 10^{12}$	$9.2 \cdot 10^{14}$
$O(n^2)$	Cuadrática	7700	290000	$5.6 \cdot 10^6$	$1.7 \cdot 10^8$
$O(n^3)$	Cúbica	391	4420	31493	315938
$O(2^n)$	Exponencial	26	36	45	55
$O(n!)$	Factorial	11	13	16	18

Tabla 9-1 Costos más comunes, sus nombres y tiempo estimado de ejecución

Como regla general es fácilmente demostrable que para dos funciones f y g se cumple:

Si existe n_0 tal que para todo $n \geq n_0$ se tiene que $f(n) \geq g(n)$ entonces:

$$f(n) + g(n) \leq 2 * f(n)$$

Por tanto

$$f(n) + g(n) = O\big(f(n)\big)$$

Se invita al lector a demostrar formalmente las siguientes identidades.

$$cf(n) = O\big(f(n)\big)$$

$$f(n) + g(n) = O\left(\max\big(f(n), g(n)\big)\right)$$

$$\log_a n = O(\log n)$$

$$\log n^k = O(\log n)$$

$$log(n!) = O(n\ log\ n)$$

$$1 + 2 + \cdots + n = O(n^2)$$

9.2 Búsqueda Secuencial en un Array

En el capítulo sobre arrays se estudió cómo se pueden recorrer todos los elementos de un array de enteros para buscar si uno de los elementos es igual a un valor específico. El método para buscar un elemento dentro de un array es el que recorre todos los elementos uno tras otro hasta encontrar el que se busca o hasta terminar el array (Ver Listado 9-1).

```
public static bool Contiene(int[] a, int x) {
    for (int i = 0; i < a.Length; i++)
        if (a[i] == x) return true;
    return false;
}
```

Listado 9-1 Método de búsqueda de un elemento en un array

La instrucción `i=0` se hace una vez al empezar y una de las dos operaciones `return true` o `return false` se hace también una vez antes de terminar. De modo que si denotamos por $t(\langle inst \rangle)$ el tiempo que demora determinada instrucción en ejecutarse, el costo de ejecutar esta parte del método es t(`i=0`) + t(`return true` o `return false`), llamemos a esto

udiantes de MATCOM UH

c_1. En cada iteración del ciclo anterior se repiten las operaciones `i<a.Length`, `a[i] == x` e `i++`, luego cada iteración del ciclo demora *t*(`i<a.Length`) + *t*(`a[i]==x`) + *t*(`i++`), llamemos a esto c_2. Los valores de c_1 y c_2 pueden ser mayor o menor en dependencia del código generado por el compilador C# y de la velocidad del procesador sobre la que el CLR realizará la ejecución del código generado, pero el tiempo que demora el algoritmo anterior es proporcional a la cantidad de veces que se repite el ciclo y esto en este caso depende de la longitud del array.

Note que el tiempo de la consulta en el array t(`a[i]`) se tomó constante; dado que los elementos de un array están ubicados de forma contigua en la memoria, demora el mismo tiempo acceder a cualquier elemento sin importa el valor del índice. Todas las operaciones básicas (aritméticas, de comparación, asignación) sobre tipos simples como int, bool, char, float, asumimos ejecutan en un tiempo constante.

La longitud del array es en este caso lo que se ha denominado como la n o volumen de entrada de datos. Note que si el elemento a buscar no está en el array el ciclo se hace todas las veces por lo que el tiempo que demora el método será (en el caso peor) $c_1 + nc_2$.

Informalmente podemos pensar que el tiempo "interesante" de calcular en un algoritmo es el que se obtiene para los peores casos a los que se enfrenta. Si lo vemos desde la perspectiva de un cliente se desea conocer cuán preparado está el algoritmo para las peores situaciones, no para las mejores. Sería un exceso de optimismo suponer que el elemento que se desea buscar estará siempre en las primeras posiciones del array.

No obstante, en muchos casos conviene realizar un análisis más riguroso de las características que tendrán las entradas a las que se enfrentará nuestro algoritmo para determinar el tiempo promedio (ser más justos con la capacidad de respuesta de nuestro algoritmo). Estos análisis con probabilidades y estadísticas escapan del ámbito de este capítulo y no serán presentados. Si no se especifica para qué escenario se está determinando el costo algorítmico, se deberá asumir cualquier posible entrada (ya sea la mejor o la peor).

Si encontrar la cota superior más ajustada del costo de un algoritmo es la manera de hablar de cuán bueno es el mismo, tendría sentido hablar también de cuán malo es. Es decir, encontrar una función que represente una cota inferior ajustada del costo del algoritmo para el peor caso. Esto se conoce como cota inferior asintótica u Omega (Ω).

Decimos que la función $f(n)$ es $\Omega(g(n))$ y se escribe $f(n) = \Omega(g(n))$ o $f(n) \in \Omega(g(n))$ si existen n_0 y c tal que para todo $n \geq n_0$ se cumple que $f(n) \geq cg(n)$ como se ilustra en la Figura 9-2.

Draft para estudiantes de

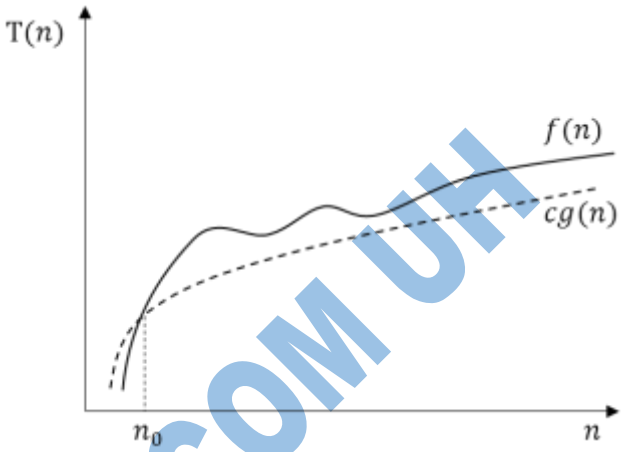


Figura 9-2 Análisis de la cota inferior asintótica

Para poder demostrar que nuestro algoritmo no puede ejecutar menos de $cg(n)$ operaciones para el peor caso, podemos aplicar una técnica conocida como **técnica del adversario**. Es decir, "acomodar" la entrada de forma tal que nuestro algoritmo esté obligado a ejecutar la mayor cantidad de instrucciones posibles.

Nuestro algoritmo recorre el array preguntando en cada posición si el elemento actual es igual al buscado, si no, continúa. ¿Qué haría el adversario? Ubicaría el elemento a buscar en la última posición o simplemente no lo ubica. Nuestro algoritmo está obligado a buscar por los n elementos del array y ello conllevaría realizar al menos n consultas. Pero el adversario puede demostrar una propuesta aún más fuerte... ¡No existe un algoritmo capaz de determinar si un elemento está o no en un array desordenado que pueda ejecutar siempre menos de n consultas!

La demostración por la técnica del adversario diría lo siguiente: suponga que existe dicho algoritmo, y que resuelve determinar si un elemento x está o no dejando de consultar una posición determinada del array... el adversario prepara la entrada poniendo valores distintos de x en todas las posiciones consultadas por el algoritmo. Si el algoritmo dice que x no está, el adversario coloca en la posición evitada del array el valor x haciendo que el algoritmo falle; si dice que sí, coloca un valor distinto de x. En otras palabras, si el algoritmo no consulta todas las posiciones, existe una posible entrada con la que falla.

Esto puede resultar una buena "justificante" para un cliente pues logramos demostrar que nuestro algoritmo es tan malo como cualquier otro.

Si encontramos que la misma función $g(n)$ representa a la vez una cota superior asintótica y una cota inferior asintótica, podemos decir que nuestro algoritmo es $\Theta(g(n))$. Por ejemplo, nuestro algoritmo de búsqueda es $\Theta(n)$ para el caso peor[1].

Decimos que la función $f(n)$ es $\Theta(g(n))$ y se escribe $f(n) = \Theta(g(n))$ o $f(n) \in \Theta(g(n))$ si existen n_0, c_1 y c_2 tal que para todo $n \geq n_0$ se cumple que $c_1 g(n) \leq f(n) \leq c_2 g(n)$ como se ilustra en la Figura 9-3.

[1] El apellido "para el caso peor" lo repetimos puesto que técnicamente, la búsqueda puede ejecutar en tiempo constante en el mejor caso.

udiantes de MATCOM UH

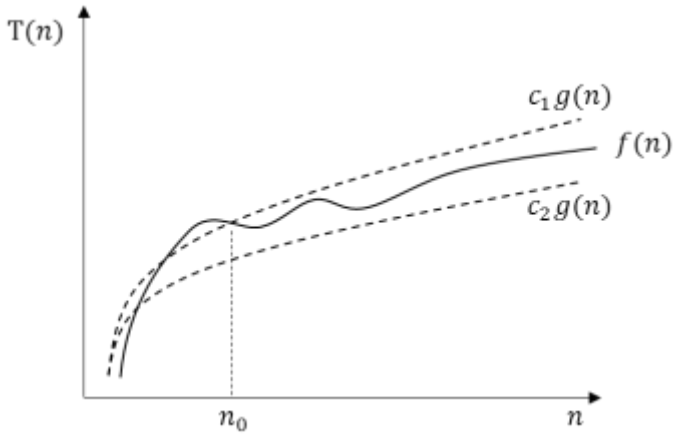


Figura 9-3 Análisis de la cota asintótica ajustada

Aunque se puede pensar que es comprensible que el tiempo de búsqueda en un array sea linealmente proporcional a la cantidad de elementos del array, para muchas aplicaciones computacionales esta operación de búsqueda es tan frecuente que puede valer la pena tratar de encontrar una manera de reducir dicho orden. Esto se verá en el epígrafe siguiente.

9.3 Búsqueda Binaria en un Array

Piense como Ud. busca el teléfono de una persona en un listín telefónico, o cómo encuentra determinada página de un libro por su número. Difícilmente Ud. encontraría el nombre si el listín no estuviese ordenado. Ud. no busca secuencialmente el nombre a través de todo el listín, ni recorre de una en una las páginas del libro para ver la deseada. Ud. aprovecha que los nombres están en orden alfabético y que las páginas están enumeradas de forma consecutiva para llegar de forma rápida al elemento deseado. El algoritmo de *búsqueda binaria* en un array se basa en esta misma idea.

Draft para estudiantes de

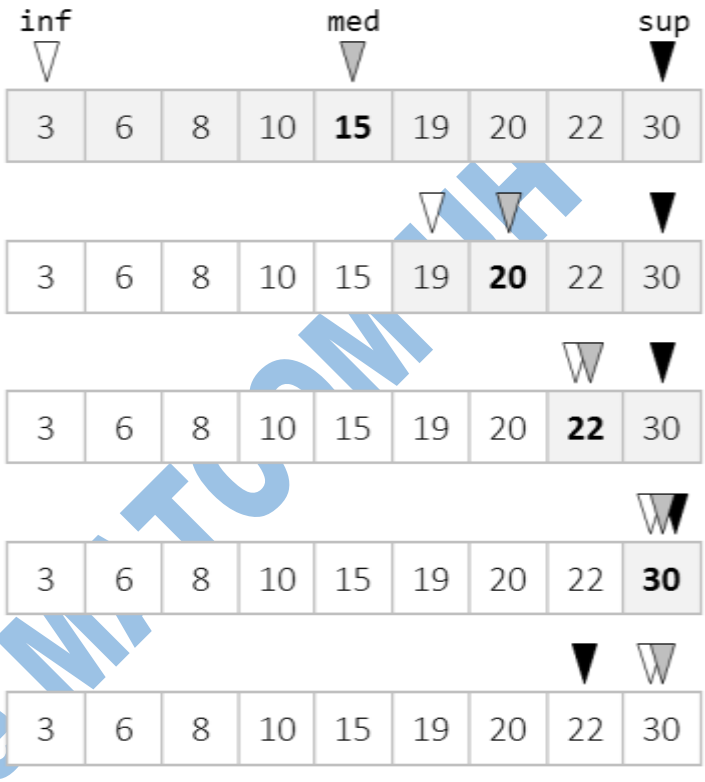


Figura 9-4 Ejecución de la búsqueda binaria del valor 24 en un array

Considere que los elementos de un array de números enteros están ordenados de menor a mayor (la idea es la misma si los elementos del array fuesen de otro tipo siempre que estén ordenados).

Sea por ejemplo un array con los elementos {3, 6, 8, 10, 15, 19, 20, 22, 30} se quiere buscar si el elemento 24 está en el array (Figura 9-4). Tomemos el elemento que está en la posición intermedia del array (15 en este ejemplo), si el elemento a buscar es igual a éste la búsqueda habrá terminado, en caso contrario, como los elementos en el array están ordenados, si el elemento a buscar es menor que el del medio entonces hay que buscar a la izquierda de éste y si es mayor a la derecha. En este caso 24 es mayor que 15 luego hay que buscar en el subarray {19, 20, 22, 30}. Con esta sección del array se repite de nuevo el mismo procedimiento. Cuando la cantidad de elementos es par se considerará, de los dos posibles, que el del medio es el más a la izquierda (un razonamiento similar podría hacerse tomando el más a la derecha) por lo que ahora el elemento del medio será 20, como 24 es mayor que 20 se deberá buscar ahora en la sección {22, 30}. Ahora el del medio es 22 y como 24 es mayor que 22 entonces la próxima sección en la que buscar es {30}. El elemento medio de ésta es el propio 30 que no es igual 24, pero ya no se pueden seguir haciendo subdivisiones luego se puede concluir que el ¡24 no está! en el array.

La implementación de este algoritmo fue vista con anterioridad en el capítulo de arrays y nuevamente abordado en el tema de recursividad. En este capítulo se analiza su costo (Listado 9-2).

```
public static bool BúsquedaBinaria(int[] a, int x) {
    int inf = 0;
    int sup = a.Length - 1;
```

Estudiantes de MATCOM

```
    while (sup >= inf) {
        int med = (inf + sup) / 2;
        if (x < a[med]) // de estar sería en la primera mitad
            sup = med - 1;
        else
        if (x > a[med]) // de estar sería en la segunda mitad
            inf = med + 1;
        else // está en el medio
            return true;
    }
    return false;
}
```

Listado 9-2 Algoritmo iterativo de Búsqueda Binaria

Note que en cada iteración del ciclo la longitud del intervalo a buscar se reduce a la mitad (de aquí el nombre de búsqueda binaria). La última iteración (en caso de que el elemento a buscar no haya sido encontrado) es cuando el intervalo tiene longitud 1 (`sup == inf`). De modo que si n es la cantidad de elementos del array (longitud del intervalo inicial). La longitud en cada paso será $n, n/2, n/4, ..., 1$. Es decir, cuántas veces hay que dividir a n entre dos para obtener 1, o sea cuál es el k tal que $\frac{N}{2^k} = 1$, la respuesta es $k = \log_2 n$.

Se puede decir entonces que el costo de la búsqueda binaria es de orden $O(\log_2 n)$ lo que mejora sustancialmente el costo de la búsqueda secuencial que era de orden lineal $O(n)$. Ciertamente en la búsqueda binaria en cada iteración del ciclo se hacen más operaciones que en la búsqueda secuencial: `(sup>=inf)`, `med=(inf+sup)/2`, `(x<a[med])`, `(x>a[med])` y `sup=med-1` o `inf=med+1`, es decir 5 operaciones contra 3 en la búsqueda secuencial. Note sin embargo, que el costo de estas operaciones sigue siendo constante (es decir no depende de n longitud del array) llamemos a esta constante c_{bin}. De modo que en la medida en que se trabaje con arrays de mayor longitud la diferencia con la búsqueda secuencial del epígrafe anterior será apreciable. Es decir aunque $c_{\text{sec}} < c_{\text{bin}}$ se puede encontrar una longitud n tal que a partir de dicho valor se cumpla que $c_{\text{bin}} * \log_2 n < c_{\text{sec}} * n$, dada que la función $T(n) = n$ crece mucho más rápido que $T(n) = \log_2 n$. Por ejemplo para n igual a 1000 el costo de la búsqueda secuencial es proporcional a 1000 mientras que el costo de la búsqueda binaria es proporcional a $\log_2 1000 \approx 10$.

Es tan importante y útil un algoritmo de búsqueda binaria que C# dispone en la clase `System.Array` diferentes métodos `BinarySearch` para tales propósitos (ver capítulo sobre arrays).

9.4 Ordenación

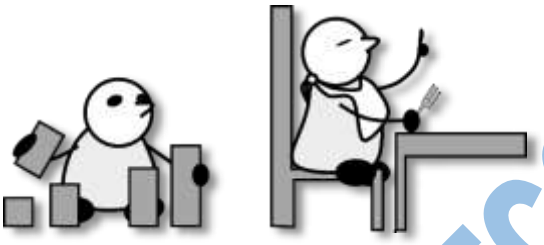
El éxito del algoritmo de búsqueda binaria se basa en que los elementos del array están ordenados. La *ordenación* es parte integrante de muchas aplicaciones de las computadoras[2]. En muchas ocasiones los resultados producidos por un determinado cómputo están ordenados de alguna manera. Muchas aplicaciones son eficientes porque se basan en que los datos que utilizan están ordenados, o porque pueden utilizar previamente algún método para ordenarlos.

[2] De hecho uno de los términos en castellano con que se conoce a los computadoras es el término *ordenador* (del francés *ordenateur*).

Draft para estudiantes de

Como se vio en el algoritmo anterior, buscar en un array ordenado es mucho más eficiente que buscar en un array no ordenado. Esto también es especialmente cierto para los humanos, ya vimos que es relativamente fácil buscar el teléfono de una persona en el listín; pero sería muy incómodo buscar una palabra en el índice de materias de este libro si este índice de materias no apareciese ordenado alfabéticamente.

La ordenación es una de las operaciones más importantes y más estudiadas en la Ciencia de la Computación. Este epígrafe no pretende, por supuesto, un estudio exhaustivo de este tema, lo cual es imposible de abarcar aquí y es objetivo de cursos y libros más específicos, sino solamente mostrar al lector algunos de los algoritmos de ordenación más sencillos, ejercitarlo en su programación en C# y en el análisis del costo de los mismos.



9.4.1 Ordenación por mínimos sucesivos

Tal vez el método de ordenación más simple es el que se conoce como *mínimos sucesivos*. Se hace un recorrido del array y cada vez que se encuentre un elemento menor que el que está en la primera posición entonces se intercambia con éste. Esto garantiza que al finalizar el recorrido habrá quedado en la primera posición el menor de todos los elementos del array. Se vuelve a repetir un proceso similar de recorrido pero ahora empezando en la segunda posición, al finalizar este segundo recorrido quedado en la segunda posición el siguiente menor. Se realiza un recorrido similar empezando en cada una de las posiciones hasta un último recorrido que será el que empiece en el penúltimo elemento. Después de hacer todos los recorridos el array habrá quedado ordenado de menor a mayor. La Figura 8.3 ilustra los diferentes cambios para ordenar un array con los elementos {4, 8, 3, 5, 2, 6}. La implementación en C# se muestra en el Listado 9-3.

```
public static void Ordenar(int[] a) {
    for (int k = 0; k < a.Length; k++)
        Intercambiar(a, k, PosDelMenor(a, k));
}
private static int PosDelMenor(int[] a, int inicio) {
    int p = inicio;
    for (int i = inicio + 1; i < a.Length; i++)
        if (a[i] < a[p])
            p = i;
    return p;
}
```

Listado 9-3 Algoritmo de ordenación por mínimos sucesivos

Estudiantes de MATCOM UH

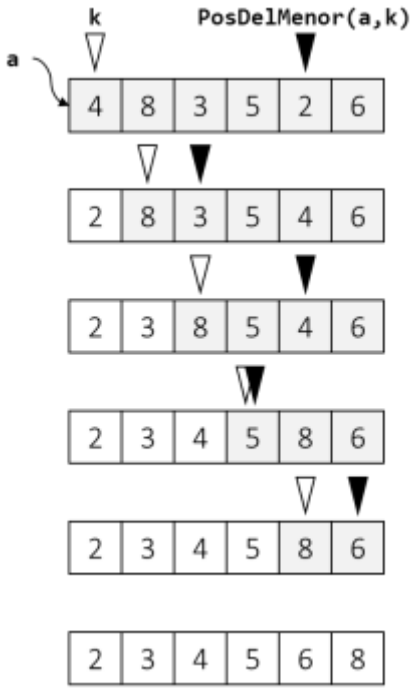


Figura 9-5 Ejemplo de ejecución del algoritmo de ordenación de mínimos sucesivos

Siendo n la longitud del array de entrada, en este algoritmo el ciclo del método `Ordenar` ejecuta $n-1$ iteraciones y llama a la función `PosDelMenor` para luego intercambiar los valores respectivos. No obstante, el método `PosDelMenor` se ejecuta con distintos parámetros y su tiempo varía. El costo global de ordenar $\mathrm{T_o}$ quedaría:

$$\mathrm{T_o}(n) = c_0 + (n-1)c_1 + \mathrm{T_p}(n,0) + \mathrm{T_p}(n,1) + \cdots + \mathrm{T_p}(n,n-1)$$

Donde c_0 es la constante de inicialización de la variable de control del ciclo, c_1 el tiempo que requieren el chequeo de la condición del ciclo, el incremento y el intercambio (que se realizan por cada iteración).

El tiempo de ejecución de `PosDelMenor` $\mathrm{T_p}$ depende del parámetro inicio (que indica el comienzo del ciclo de búsqueda del menor) e itera por `n - inicio` elementos por tanto el tiempo de ejecución:

$$\mathrm{T_p}(n, inicio) = c_2 + c_3(n - inicio)$$

Donde c_2 es la constante de inicializar las variables `p` e `i`, además del retorno; c_3 es la constante de la ejecución de cada iteración del ciclo (que incluye condición de parada, incremento, la condición `a[i] < a[p]` y en el peor caso una asignación en cada iteración.

Sustituyendo nos queda un tiempo total para el algoritmo de ordenación:

$$\mathrm{T_o}(n) = c_0 + (n-1)c_1$$
$$+\big(c_2 + c_3(n-0)\big) + \big(c_2 + c_3(n-1)\big) + \cdots + \Big(c_2 + c_3\big(n-(n-1)\big)\Big)$$

$$\mathrm{T_o} = c_0 + (n-1)c_1 + nc_2 + c_3\frac{n(n+1)}{2}$$

Draft para estudiantes de

$$T_o = (c_0 - c_1) + n\left(c_1 + c_2 + \frac{c_3}{2}\right) + n^2\left(\frac{c_3}{2}\right)$$

$$T_o = c_a + nc_b + n^2c_c$$

En esta expresión la función que crece más rápido es la cuadrática (n^2c_c) por lo que se dice que el tiempo algorítmico es $O(n^2)$.

Para la ejecución de este algoritmo no importa la naturaleza de la entrada, siempre tiene el mismo costo algorítmico. Sería preferible que en las ocasiones en que la entrada esté "casi" ordenada, el costo disminuya. Analicemos a continuación un algoritmo de ordenación que intuitivamente "aproveche" entradas casi ordenadas.

9.4.2 Ordenación por burbuja o por comparación de pares consecutivos

El método de ordenación conocido como ordenación burbuja (*bubble sort*) consiste en recorrer todos los elementos del array comparando cada elemento con el siguiente e intercambiarlos si el siguiente es menor de ahí también el nombre de ordenación por comparación de pares consecutivos. Este proceso de recorrido se repite hasta que en un recorrido no se hayan hecho intercambios, lo que significa que ya habrán quedado ordenados. La ordenación del array {4, 8, 3, 5, 2, 6} se ilustra en la Figura 9-6. En gris están señalados los pares que se chequean, y resaltados con el borde negro las inversiones que son "corregidas".

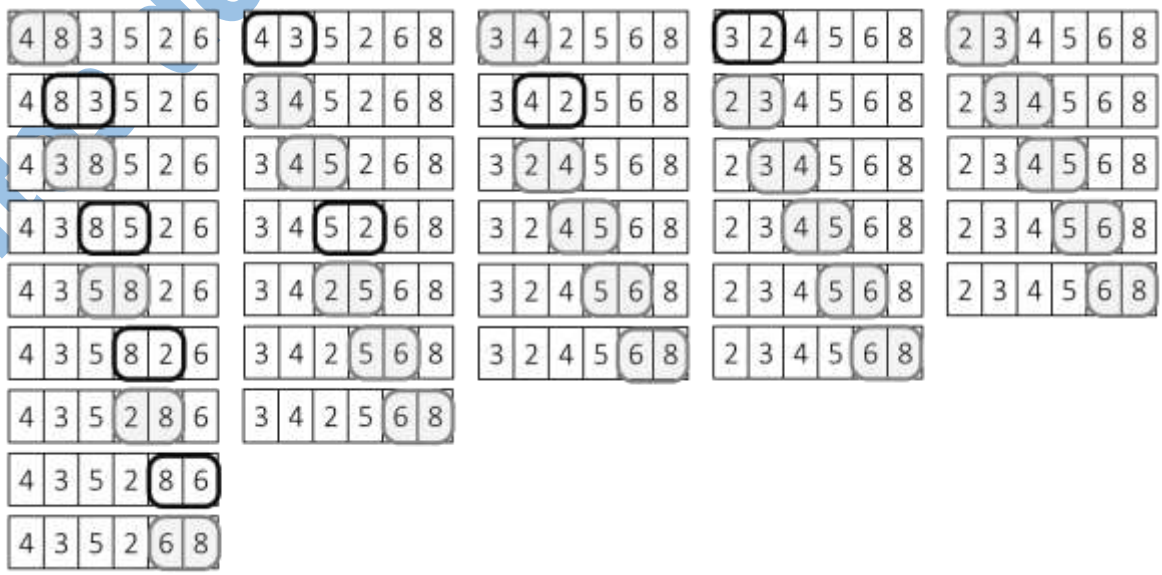


Figura 9-6 Ejecución del algoritmo de ordenación por burbuja

Un posible código para este algoritmo se presenta en el Listado 9-4.

```
public static void OrdenaPorBurbuja(int[] a) {
    bool huboIntercambio;
    do {
        huboIntercambio = false;
        for (int i = 0; i < a.Length - 1; i++)
            if (a[i] < a[i + 1]) {
                Intercambia(a, i, i + 1);
                huboIntercambio = true;
```

Estudiante de MATCOM UH

```
            }
        } while (huboIntercambio);
    }
```

Listado 9-4 Algoritmo de Ordenación por Burbuja

El ciclo `for` tendrá un costo lineal con respecto a la cantidad de elementos del array (n). Sólo es necesario conocer cuántas veces se ejecuta el mismo en el caso peor. Si analizamos este algoritmo podemos ver que con la primera "pasada" se garantiza que el mayor valor queda ubicado en la última posición, con la segunda, que el segundo mayor valor quede en la penúltima y así sucesivamente. Es decir, no se requerirán más de $n-1$ pasadas. Suponiendo que se ejecutan k pasadas, el costo quedaría $c_0 + (c_1 n + c_2)k = c_0 + c_2 k + c_1 nk = O(nk)$. Como k puede ser $n-1$ en el peor caso, el algoritmo quedaría $O(n^2)$.

No obstante, para los casos en los que el array tiene los elementos casi ordenados y solo haya necesidad de un número constante k de pasadas, el algoritmo tiene un costo lineal.

Este algoritmo puede ser mejorado si consideramos el hecho de que si en una pasada, a partir de cierta posición no ocurren más intercambios, entonces los elementos posteriores a dicha posición están ordenados y en su posición final.

Para demostrar esto trataremos primero de demostrar una invariante de ciclo en el algoritmo de burbuja.

"Si ocurre un intercambio entre la posición i y la $i+1$, el elemento que queda en $i+1$ es el mayor de todos los elementos del array hasta dicha posición". Supongamos que no, que existe una posición $j < i+1$ que tiene el mayor valor del array en el rango $0 \ldots i+1$. El algoritmo dejó dicho valor en esa posición porque el siguiente era mayor (si no hubiese realizado un intercambio) pero esto contradice que en j esté el mayor valor del array en dicho rango.

Volviendo a nuestra proposición inicial, si el último intercambio ocurren entre la posición t y $t+1$, entonces se cumple: primero, que los elementos desde la posición $t+1$ hasta el final están ordenados; segundo, como el elemento de la posición $t+1$ es mayor que los anteriores, entonces los elementos desde la posición $t+1$ hasta el último están ubicados en su posición final (no necesitarán más cambios). Con este resultado podemos hacer una modificación al algoritmo que nos permita acotar el ciclo for que realiza los intercambios evitando que llegue al final innecesariamente. Esta modificación puede ser vista en el código del Listado 9-5.

```
public static void OrdenaPorBurbujaMejorado(int[] a) {
    int posUltIntercambio = a.Length;
    do {
        int ultPosAIntercambiar = posUltIntercambio - 1;
        posUltIntercambio = -1;
        for (int i = 0; i < ultPosAIntercambiar; i++)
            if (a[i] < a[i + 1]) {
                Intercambia(a, i, i + 1);
                posUltIntercambio = i;
            }
    } while (posUltIntercambio != -1);
}
```

Listado 9-5 Algoritmo de Ordenación por Burbuja Mejorado

Draft para estudiantes de

El cambio en este código garantiza poder conocer entre una pasada y la siguiente, la posición del último intercambio realizado. Con esta variable se puede expresar también el hecho de que se haya realizado un cambio (valor distinto de -1). Como en la primera pasada no hay "cambios anteriores" entonces se inicializa con el valor `a.Length` para forzar a que se recorra todo el array en ese primer recorrido.

¿Cuál es el peor escenario para este algoritmo mejorado? ¿Cambia el orden de nuestro algoritmo de ordenación con esta mejora? Dejamos estas preguntas para su consideración.

Aunque los algoritmos de ordenación analizados tienen costos $O(n^2)$, existen otros que mejoran este tiempo a $O(n\sqrt{n})$, $O(n \log n)$ e incluso ¡$O(n)$! No obstante, para los algoritmos de ordenación basados en comparaciones, existe una cota inferior asintótica demostrada $\Omega(n \log n)$.

9.5 Analizando el costo de funciones recursivas

En el capítulo de recursividad se analizaron dos estrategias diferentes de divide y vencerás que resolvían el problema de la ordenación. El método de ordenación por mezcla (*merge sort*) y el *quicksort*.

Ambos algoritmos particionan el problema en 2 subarrays, invocan recursivamente en ambos subarrays y realizan una operación de orden lineal sobre los elementos del array (en el caso del *merge-sort* la mezcla y en el caso del *quicksort* la función particiona).

Analicemos detalladamente estas funciones. El Listado 9-6 muestra el algoritmo de mezcla. Note que en cada iteración del ciclo se avanza uno de los dos índices `izq` o `der`. La cantidad de veces que se pueden avanzar estos índices en total no supera n (el tamaño del subarray mezclado[3]). Si quedan valores por ubicar (porque se termina algunas de las mitades durante la mezcla), son colocados en el array `aux` mediante `Array.Copy` para completar los n elementos a dejar ordenados en el array aux. Al final se copian nuevamente los n elementos desde el array `aux` al array original `a`.

```
private static void Mezclar(int[] a, int[] aux, int inf, int medio, int sup) {
  int izq = inf; // para recorrer la 1ra mitad
  int der = medio + 1; // para recorrer la 2da mitad
  int pos = inf; // para ir dejando la mezcla en aux
  while (izq <= medio && der <= sup) {
    if (a[izq] < a[der]) aux[pos++] = a[izq++];
    else aux[pos++] = a[der++];
  }
  if (izq <= medio) Array.Copy(a, izq, aux, pos, medio - izq + 1);
  if (der <= sup) Array.Copy(a, der, aux, pos, sup - der + 1);
  Array.Copy(aux, inf, a, inf, sup - inf + 1);
}
```

Listado 9-6 Algoritmo de Mezcla

El costo de este algoritmo se puede calcular con la siguiente expresión:

$$T_m(n) = c_0 + c_1 a + c_2 b + c_3 n$$

[3] Para un subproblema en particular, el costo habrá que analizarlo dependiendo del tamaño de dicho subproblema, no del problema original que se esté descomponiendo. En este ejemplo, del array original se está procesando un subarray de tamaño sup-inf+1, no a.Length.

Estudiantes de MATCOM UH

Donde a representa los valores copiados durante el `while` y b los valores copiados posteriormente con `Array.Copy`. El término $c_3 n$ representa el costo de reubicar nuevamente los valores hacia el array original. Como $a + b = n$, el costo queda finalmente:

$$\mathrm{T_m}(n) = c_0 + c_1 a + c_2 b + c_3 n \leq c_0 + \max(c_1, c_2)\, n + c_3 n$$
$$= c_0 + c' n = O(n)$$

En el caso de la función `Particiona` en el quicksort (Listado 9-7) se puede calcular el costo de forma similar.

```
static int Particiona(int[] a, int ini, int fin, int pivote) {
    int i = ini - 1;
    int j = fin + 1;
    while (true) {
        do i++; while (a[i] < pivote);
        do j--; while (a[j] > pivote);
        if (i >= j)
            return j;
        var temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

Listado 9-7 Método para particionar en el algoritmo QuickSort

Recordando; este algoritmo recorre el array de izquierda a derecha con `i` y de derecha a izquierda con `j`, intercambiando si procede un elemento en `i` mayor que el pivote, con un elemento en `j` menor que el pivote. Si se "cruzan" los índices se concluye el particionamiento.

Sea a la cantidad de veces que se hace `i++` y b la cantidad de veces que se ejecuta `j--` antes de cruzarse. En cada iteración del `while` externo se incrementa al menos en 1 el valor de i y se decrementa al menos en 1 el valor de j, por tanto no se harán más de $n/2$ iteraciones. El costo puede calcularse como:

$$\mathrm{T_p}(n) = c_0 + a c_1 + b c_2 + c_3 \frac{n}{2}$$

Donde c_0 es el costo de las inicializaciones y del retorno. El valor c_1 y c_2 es el costo de las condiciones de los `while` internos. El costo de la comparación `i>=j` y las instrucciones de intercambio se representan con la constante c_3.

Como $a + b = n$ y el tiempo de las comparaciones `a[i]<pivote` y `a[i]>pivote` son el mismo se tiene:

$$\mathrm{T_p}(n) = c_0 + c_4 n + c_3 \frac{n}{2} = O(n)$$

¿Cómo podemos considerar estos tiempos en el análisis de un algoritmo recursivo? Detallemos primeramente el algoritmo de ordenación por mezcla (Listado 9-8).

```
private static void OrdenarPorMezcla(int[] a, int[] aux, int inf, int sup) {
  if (inf < sup) {
    int medio = (inf + sup) / 2;
    OrdenarPorMezcla(a, aux, inf, medio);
    OrdenarPorMezcla(a, aux, medio + 1, sup);
```

Libro para estudiantes de

```
    Mezclar(a, aux, inf, medio, sup);
  }
}
```

Listado 9-8 Método de ordenación por mezclas

En este caso podemos analizar el costo mediante una expresión ¡recursiva!

$$\mathrm{T_o}(n) = \begin{cases} c_0 & n = 1 \\ c_0 + \mathrm{T_o}\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + \mathrm{T_o}\left(\left\lceil\frac{n}{2}\right\rceil\right) + \mathrm{T_m}(n) & n > 1 \end{cases}$$

Informalmente vamos a trabajar con $n/2$ en lugar de tratar rigurosamente los tamaños con parte entera superior e inferior, asumiendo siempre un valor de n que es potencia de dos.

En estos casos el tiempo de ordenación con merge sort quedaría:

$$\mathrm{T_o}(n) = \begin{cases} c_0 & n = 1 \\ c_0 + 2\mathrm{T_o}\left(\frac{n}{2}\right) + \mathrm{T_m}(n) & n > 1 \end{cases}$$

Como nos interesa acotar superiormente el costo podemos reducir la complejidad de esta fórmula si utilizamos una constante c con que cumpla que $c_0 + \mathrm{T_m}(n) \leq cn$. Esto es posible puesto que $\mathrm{T_m}(n) = O(n)$. Quedando:

$$\mathrm{T_o}(n) \leq 2\mathrm{T_o}\left(\frac{n}{2}\right) + cn$$

Desarrollemos el $\mathrm{T_o}$ interno par de niveles más de profundidad, se tiene:

$$\mathrm{T_o}(n) \leq 2\left(2\mathrm{T_o}\left(\frac{\frac{n}{2}}{2}\right) + c\left(\frac{n}{2}\right)\right) + cn = 4\mathrm{T_o}\left(\frac{n}{4}\right) + 2cn$$

$$\mathrm{T_o}(n) \leq 4\left(2\mathrm{T_o}\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)\right) + 2cn = 8\mathrm{T_o}\left(\frac{n}{8}\right) + 3cn$$

Suponiendo que $n = 2^k$, (o lo que es equivalente $k = \log_2 n$) se tiene un desarrollo como sigue:

$$\mathrm{T_o} \leq 2^k\mathrm{T_o}\left(\frac{n}{2^k}\right) + kcn$$

Sustituyendo k queda:

$$\mathrm{T_o}(n) \leq 2^{\log_2 n}c_0 + (\log_2 n)cn$$

$$\mathrm{T_o}(n) \leq c_0 n + cn\log_2 n$$

Quedando finalmente una componente lineal y una n logaritmo de n que crece más rápido. Por tanto, el costo de la ordenación por mezcla quedaría:

$$\mathrm{T_o}(n) = O(n\log n)$$

¿Qué sucede entonces con los valores de n que no son potencia de dos?

Entre los valores n y $2n$ hay siempre una potencia de dos (llamémosle n_p). Por tanto se puede decir que:

Estudiantes de MATCOM UH

$$T_o(n) \leq T_o(n_p \leq 2n)$$

$$T_o(n) \leq T_o(n_p) \leq c' n_p \log n_p \leq c' 2n \log 2n$$

Como para $n \geq 2$ se cumple que $2n \leq n^2$, se tiene:

$$T_o(n) \leq c' 2n \log n^2 \leq c' 4n \log n \leq c'' n \log n = O(n \log n)$$

El costo del merge-sort, podría analizarse gráficamente como se muestra en la Figura 9-7.

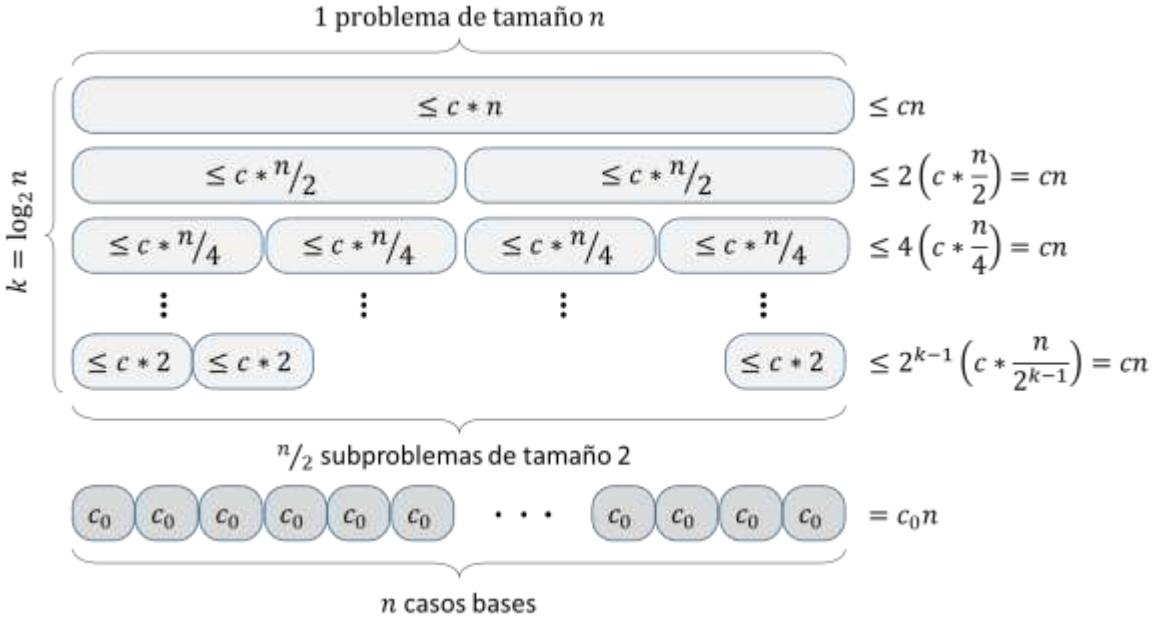


Figura 9-7 Análisis gráfico del costo algorítmico de la ordenación por mezcla

El gráfico muestra cada uno de los subproblemas en los que se descompone el algoritmo de ordenación por mezcla y cada uno de los costos locales. A la derecha se muestran las sumas totales de cada nivel de la recursión. La suma global de todas las expresiones quedaría:

$$T_o(n) \leq kcn + c_0 n$$

$$T_o(n) = O(n \log n)$$

El costo del algoritmo Quicksort puede calcularse de igual forma para el caso mejor en que todas las ejecuciones del particiona dividiera el problema exactamente a la mitad. No obstante para el caso en que para cada partición quede un elemento de un lado y el resto de los valores en el otro, se obtiene el peor tiempo.

En efecto, si suponemos que todas las particiones creadas separan el problema en 1 y $n-1$, se tiene:

$$T_q(n) = c_0 + c_1 n + T_q(1) + T_q(n-1)$$

Donde c_0 es el costo constante de las inicializaciones, etc.; $c_1 n$ es el costo lineal de la función `Particiona` y los valores $T_q(1)$ y $T_q(n-1)$ los costos de las llamadas recursivas a cada subarray del particionamiento.

Draft para estudiantes de

Sustituyendo $T_q(1) = c_0$ y suponiendo una constante c tal que $c_0 + c_1 n \leq cn$ se obtiene

$$T_q(n) \leq T_q(1) + T_q(n-1) + cn = c_0 + T_q(n-1) + cn$$

Desarrollando esta expresión tenemos:

$$T_q(n) \leq c_0 + \Big(c_0 + T_q(n-2) + c(n-1)\Big) + cn = 2c_0 + T_q(n-2) + c(n+n-1)$$

$$T_q(n) \leq 3c_0 + T_q(n-3) + c(n+n-1+n-2)$$

Generalizando:

$$T_q(n) \leq kc_0 + T_q(n-k) + c(n+n-1+\cdots+n-k+1)$$

El caso de parada se alcanza cuando $n-k=1$, es decir, para $k=n-1$. La expresión del costo queda:

$$T_q(n) \leq (n-1)c_0 + c_0 + \left(\frac{n(n+1)}{2} - 1\right)c = n^2 c_a + nc_b + c_c = O(n^2)$$

Entonces tenemos que el algoritmo Quicksort es para el caso peor ¡$O(n^2)$!

La Figura 9-8 muestra gráficamente este fenómeno y un análisis equivalente del costo global del algoritmo para este caso peor.

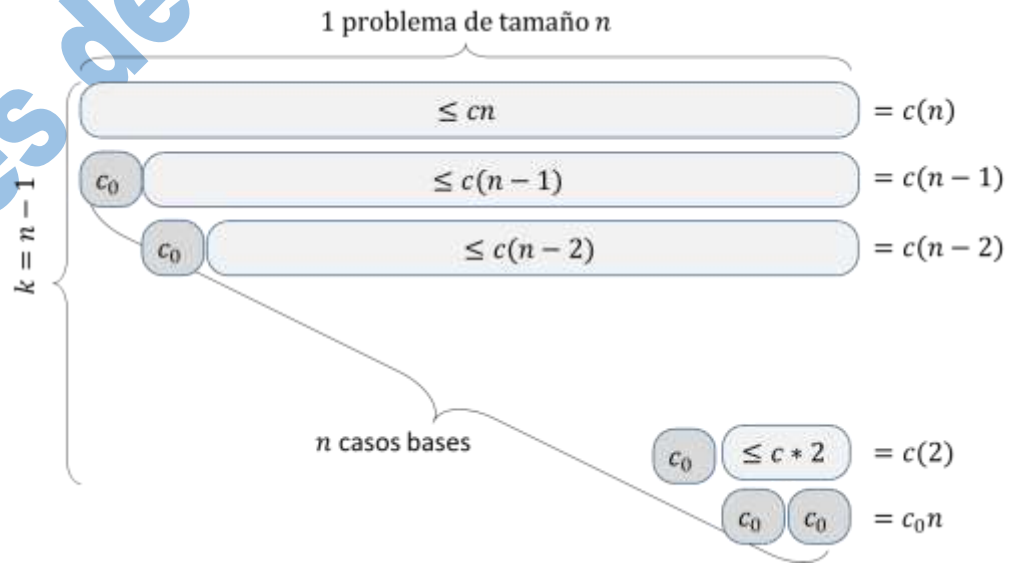


Figura 9-8 Análisis gráfico del costo algorítmico de la ordenación con Quicksort para el peor caso

¿Qué sucede si lo común es que la partición queda siempre de la forma an y $(1-a)n$, con $0 < a < 1$?

El mejor valor para a es $1/2$, es decir, particionar a partes iguales. Lo peor sería acercarse a 0 o a 1, representando el particionamiento que deja casi todos los elementos de un lado y el resto en el otro subarray. Sin embargo, cualquier otro valor de a (1/3, 2/3, 1/4, 1/5 ...) es aceptable! Veamos la demostración.

La relación recurrente para el análisis del costo quedaría:

$$T_q(n) \leq T_q(an) + T_q\big((1-a)n\big) + cn$$

Estudiantes de MATCOM UH

Supongamos sin perder generalidad que $a \geq 1 - a$; podemos decir que en el peor caso la profundidad de las llamadas recursivas será un k, tal que:

$$a^k n = 1$$

Para cada nivel de recursión no se excederá de un costo total menor o igual a un cn.

Para que $a^k n = 1$ tiene que cumplirse $k = \log_{1/a} n$.

Quedando un costo total no superior a $cn \log_{1/a} n$; nuevamente ¡$O(n \log n)$!

La selección del pivote en el *QuickSort* es determinante en la eficiencia de este algoritmo. Por ejemplo, la selección del primer elemento como pivote provoca que las entradas ordenadas sean el peor caso de ejecución (dejamos al lector su análisis). En estos casos, escoger como pivote el elemento del medio puede resultar beneficioso porque particionaría parejo el array en partes de similar tamaño.

Otras estrategias usadas comúnmente son:

- Escoger un elemento aleatorio del array como pivote.
- Escoger la mediana entre primer, medio y último elementos.

Aun cuando la mediana de todo el array se puede calcular con un costo lineal (lo cual garantizaría una ejecución $O(n \log n)$ para el caso peor) no se utiliza puesto que incrementa considerablemente la constante del costo.

Para valores de n pequeños (por debajo de 40) es preferible algoritmos $O(n^2)$ con una constante baja como burbuja u ordenación por inserción. Una buena técnica es emplear *QuickSort* para valores superiores del tamaño del array y concluir con uno de estos algoritmos para tamaños pequeños.

Este tipo de análisis en algoritmos recursivos puede ser una herramienta muy útil para determinar el costo computacional de una solución.

Se invitamos al lector que determine para cada uno de los siguientes casos, un problema real para el cual dicha expresión represente su costo, y el orden del mismo.

Ejercicio 1.

$$T(n) = \begin{cases} c_0 & n = 1 \\ T\left(\frac{n}{2}\right) + c_1 & n > 1 \end{cases}$$

Ejercicio 2.

$$T(n) = \begin{cases} c_0 & n = 1 \\ 2T\left(\frac{n}{2}\right) + c_1 & n > 1 \end{cases}$$

Ejercicio 3.

$$T(n) = \begin{cases} c_0 & n = 1 \\ T(n-1) + c_1 n & n > 1 \end{cases}$$

Draft para estudiantes de

La generalización del cálculo de costos para funciones de la forma:

$$T(n) = \begin{cases} c_0 & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Se conoce como el **Teorema Maestro**. Su demostración y aplicabilidad puede encontrarse en libros especializados de diseño y análisis de algoritmos.

9.6 ¡El costo de la explosión!

Hasta ahora hemos analizado costos de funciones recursivas con estrategias divide y vencerás, pero... ¿cómo propondría usted calcular el costo de un algoritmo *backtrack*? Lo que se trata de hacer es encontrar un número que supere el costo total de las operaciones realizadas durante una búsqueda con *backtrack* para el peor caso.

Intuitivamente este número tiene relación con la cantidad de llamadas que se realizan durante el *backtracking* ($N_f(n)$) y el costo máximo de las instrucciones locales necesarias en cada llamada ($T_L(n)$).

$$T(n) \leq N_f(n) * T_L(n)$$

La cantidad total de llamadas que se realizan durante una búsqueda con Backtracking es difícil de conocer, puesto que depende de las estrategias de podas que se implementen y la entrada específica. No obstante, una buena cota superior para este número lo representa la cantidad de llamadas máximas que pueden terminar como un caso base, multiplicado por la máxima profundidad del algoritmo recursivo, conocido también en muchos casos (Ver Figura 9-9).

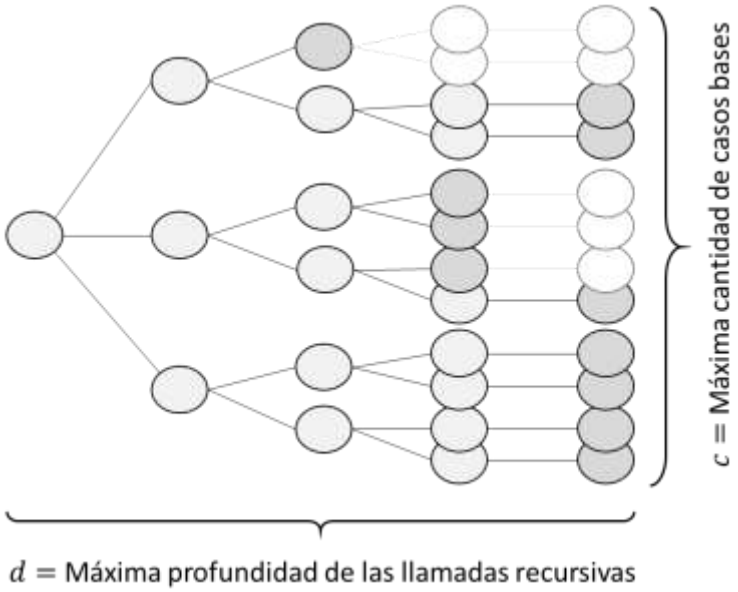


Figura 9-9 Cota superior para el número de llamadas recursivas en la ejecución de un Backtracking

Como se observa en la figura, una buena cota superior para $N_f(n)$ es dc (pueden existir al menos d llamadas para cada uno de los c casos de parada).

Estudiantes de MATCOM UH

Analicemos la solución al problema del viajante, vista anteriormente en el capítulo de recursividad. Supongamos que la entrada representa la interconexión entre n ciudades, es decir, una matriz cuadrada de n^2 elementos (formalmente la entrada es de tamaño n^2 pero para simplificar la expresión del costo trataremos la cantidad de ciudades como la "complejidad" de la entrada).

La primera ciudad a visitar puede ser cualquiera de las n posibles; la segunda sería cualquiera de las $n-1$ ciudades restantes que no haya sido visitada, y así sucesivamente hasta que se escoge la última ciudad. Es decir, se puede esperar tener aproximadamente $n(n-1)\ldots*1=n!$ casos de parada; en una búsqueda backtracking que no supera las n llamadas en profundidad (sólo se visitan n ciudades).

Una cota superior para el costo del Backtracking en este problema del viajante podría ser:

$$T(n) \leq N_f(n) * T_L(n) = n * (n!) * T_L(n)$$

Teniendo el costo local $T_L(n) \leq cn$ (el costo de buscar una ciudad que no haya sido visitada) se tiene:

$$T(n) \leq n * (n!) * cn$$

$$T(n) \leq cn^2 n!$$

Este tiempo tiene una componente polinomial ¡multiplicada por una factorial! La función factorial (o una exponencial) crece mucho más rápido que cualquier función polinomial. Por tanto, el costo polinomial es despreciable y se dice entonces que el costo es factorial (o exponencial).

El costo de la solución del viajante es $O(n!)$, pero ¿se podría tener una mejor cota? Es decir, cuando le decimos a un cliente que el algoritmo que diseñamos es $O(g(n))$ le estamos dando cierta "seguridad" de que el comportamiento nunca será peor que $g(n)$, pero es que en costos de ejecución nada es peor que un costo exponencial o factorial![4] Por tanto, no estamos diciendo mucho. Para ser justos deberíamos demostrar que nuestro algoritmo no puede ser mejor. Para ello es necesario demostrar que existe una función $g(n)$ que está siempre por debajo de nuestro costo.

¿Qué tendría que hacer el adversario para demostrar que nuestro algoritmo para el problema del viajante tiene obligatoriamente, en su peor caso, un costo factorial?

Ante todo deberá garantizar que entre todo par de ciudades exista un camino, de forma tal que no exista forma de "podar" porque no se pueda ir de una ciudad a otra. El otro aspecto es que toda permutación válida para el viajante debería costar lo mismo, de esta forma nuestro algoritmo no podría podar bajo el criterio de que se tiene un trayecto actual más costoso que uno anterior. Para ello basta asignar 1 al costo de viajar entre cualquier par de ciudades (todos los caminos cuestan n).

[4] Los problemas para los cuales no existen algoritmos con costos polinomiales se conocen como "intratables" por el rápido crecimiento de sus funciones de costo. Por ejemplo, ejecutar nuestra solución del problema del viajante en la mejor computadora de hoy día para más de 30 ciudades tomaría más tiempo que el que ha tenido la humanidad.

Draft para estudiantes de

9.7 Reduciendo la complejidad

En ocasiones concebimos una solución con costo exponencial para un problema que no lo requiere. La expresividad y legibilidad que nos brinda una solución recursiva suele esconder bajo el tapete un uso ineficiente del procesamiento y de la memoria. Esto no significa que por defecto una solución recursiva sea más ineficiente, si no que se deja de hacer un análisis más exhaustivo para dar una solución más eficiente al problema. En este epígrafe se verá cómo reconocer este tipo de problemas y cómo reducir su costo.

Analicemos el problema de encontrar el n-ésimo término de Fibonacci en su versión recursiva (Listado 9-9).

```
static long Fibonacci(int n) {
  if (n <= 2)
    return 1;
  return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Listado 9-9 Método recursivo para calcular el enésimo elemento de la sucesión de Fibonacci

¿Qué tiene de malo una solución como esta?

En términos de legibilidad, es una forma directa en que se puede traducir a código C# la formulación matemática de la sucesión:

$$F_1 = 1$$
$$F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

No obstante, el lector puede percatarse que durante la ejecución de dicho algoritmo recursivo, existen subproblemas que son analizados varias veces (Figura 9-10).

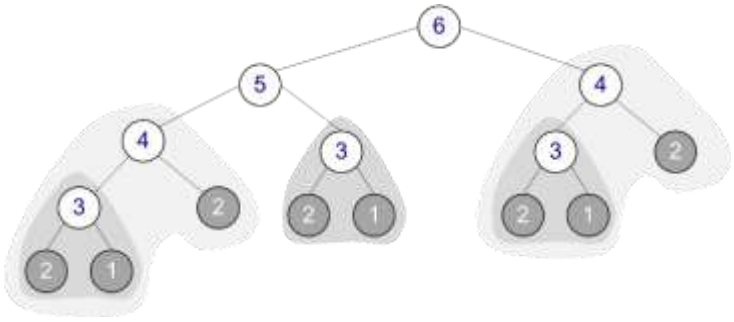


Figura 9-10 Ejecución del cálculo de Fib(6).

Una técnica muy sencilla de implementar y que reduce considerablemente el costo del algoritmo se conoce como *memoization*.

La idea es tener una tabla[5] durante la ejecución de nuestro algoritmo que guarde en la memoria, los valores ya calculados para cada subproblema. De esta forma, si durante la ejecución se vuelve a necesitar el valor para dicho subproblema, se devuelve directamente sin tener que volver a procesar. El código de Fibonacci con *memoization* puede verse en

[5] En próximos capítulos se verán diversas estructuras de datos que pueden ser provechosas en este sentido como las listas, las colas y los diccionarios. Los ejemplos de este capítulo se basarán en arrays multidimensionales.

udiantes de MATCOM UH

el Listado 9-10. Note que en este caso se utilizó un array de longitud $n + 1$ para almacenar los valores calculados de la sucesión desde el 1 hasta el n.

```
public static int Fib(int n) {
    return FibConMem(n, new int[n + 1]);
}

private static int FibConMem(int n, int[] memo) {
    if (memo[n] == 0) // no se ha calculado el valor
        if (n <= 2)
            memo[n] = 1;
        else
            memo[n] = FibConMem(n - 1, memo) + FibConMem(n - 2, memo);
    return memo[n];
}
```

Listado 9-10 Implementación de Fibonacci con Memoization

¿De qué orden con respecto a n, queda esta versión del algoritmo que determina el n-ésimo valor de la sucesión de Fibonacci?

Este costo debe ser analizado de forma global, es decir, contando cuantas veces se ejecuta cada instrucción para la ejecución completa del algoritmo.

La primera instrucción `if (memo[n] == 0)` y la última `return memo[n]` se ejecutan tantas veces se invoque el método `FibConMem`.

El `if` anidado se ejecuta una única vez por cada valor de n (para nuestro caso, todos los subproblemas $n, n-1, n-2$, ... hasta 1). Para cualquier `k`, la segunda vez en la ejecución que se consulta `memo[k]` ya tiene un valor distinto de 0 y por lo tanto la primera condición falla. Siendo así, las llamadas recursivas `FibConMem` con `n-1` y con `n-2`, no se realizarán más de n veces, por tanto todas las instrucciones se ejecutan menos de $2n$ veces, quedando $T(n) = O(n)$.

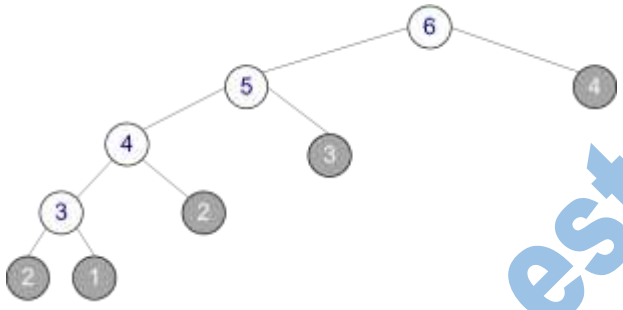


Figura 9-11 Ejecución de la funcion FibConMem(6)

La Figura 9-11 muestra el comportamiento de la ejecución de `FibConMem` y cómo convierte las restantes llamadas a subproblemas en "casos de parada" utilizando el valor almacenado en memoria.

El próximo paso sería prescindir de la recursión para evitar siempre que se pueda, el uso innecesario de la memoria de la pila. Como se observa en la figura, si se invocaran las llamadas en el orden `FibConMem(1)`, `FibConMem(2)`, ..., `FibConMem(n-1)`, `FibConMem(n)`, resultaría que los valores de los que se dependen ¡siempre están almacenados! Generalizando, sólo sería necesario resolver los subproblemas de los que se depende en el

Draft para estudiantes de

orden adecuado y depender directamente de los valores almacenados en la tabla en lugar de en llamadas recursivas. Este orden de procesamiento se conoce como ***Bottom-Up*** (de abajo hacia arriba, de lo simple a lo complejo, de lo básico a lo general) en contraste con el enfoque tradicional de un algoritmo divide y vencerás conocido por ***Top-Down***. El cálculo de los valores óptimos para los subproblemas de forma *bottom-up* es la idea principal detrás de la **programación dinámica** (*Dynamic Programming*).

El cálculo del n-ésimo valor de Fibonacci con este cambio se ilustra en el Figura 9-12.

```
static int FibDyn(int n) {
    int[] mem = new int[n + 1];
    for (int i = 1; i <= n; i++)
        if (i <= 2)
            mem[i] = 1;
        else
            mem[i] = mem[i - 1] + mem[i - 2];
    return mem[n];
}
```

Figura 9-12 Cálculo de Fibonacci con Programación Dinámica

Como de la tabla sólo se necesitan los últimos dos valores calculados podemos reducir el algoritmo a utilizar dos variables (almacenando el último y el penúltimo valor), obteniendo el algoritmo propuesto en el capítulo de ciclos.

9.7.1 Programación Dinámica

Con la programación dinámica podemos sacrificar memoria en aras de reducir procesamiento. ¿Es siempre válido este sacrificio? En un algoritmo divide y vencerás como el ordenamiento por mezclas, los subproblemas son independientes uno del otro y por tanto no existen "sub-subproblemas" comunes. En este caso, el uso de un acercamiento con *memoization* o en general, con programación dinámica, no resuelve reducir el costo ni el uso de memoria.

Por otra parte se debe pensar realmente de cuántos subproblemas depende un problema, lo que no siempre es un cálculo sencillo. Por ejemplo: en el caso del viajante… ¿diría usted que si el problema es un camino por las n ciudades un subproblema es cualquier posible subconjunto de $n-1$ ciudades? Incluso si se pudiera representar cada subproblema como un subconjunto de ciudades, esto significaría 2^n posibles subproblemas (porque hay 2^n posibles subconjuntos de $\{1,\dots,n\}$), y por lo tanto un costo exponencial de memoria y procesamiento.

Si tratamos de reducir el número de subproblemas (por ejemplo, decir que el problema es determinar el camino en las ciudades $\{1 \dots n\}$ y el subproblema es determinar el camino óptimo para las ciudades del conjunto $\{1 \dots n-1\}$), se incumple un requisito de la programación dinámica: la **subestructura óptima**. El óptimo para nuestro problema debe depender del óptimo de nuestros subproblemas. En nuestro caso, el camino óptimo para las $n-1$ primeras ciudades no tiene por qué ser válido una vez que se desea insertar la ciudad n en el recorrido.

¿Cuál sería realmente un subproblema en el viajante que tenga realmente subestructura óptima? Dejamos al lector este análisis.

udiantes de MATCOM UH

Afortunadamente, existen un gran número de problemas que pueden aprovechar la programación dinámica para reducir su costo. En este epígrafe mostraremos cómo resolver por programación dinámica dos de ellos: el cambio en monedas y determinar la mayor subsecuencia palíndromo en una cadena.

El cambio de monedas

¿Le gusta a Ud. cargar con muchas monedas en el bolsillo? ¿No se disgustaría si un cambio de 41 centavos se lo entregan en monedas de 1 centavo? El problema conocido como *dar cambio en monedas* intenta dar el cambio utilizando la menor cantidad de monedas posibles.

Muchos países utilizan monedas de 1, 5, 10 y 25 céntimos de la unidad monetaria del país (dólar, peso, real, euro, etc.). Si se dispusiera de una cantidad "ilimitada" de cada una de las monedas[6] la menor cantidad de monedas para dar un cambio de 41 centavos sería una moneda de 25, otra de 10, otra de 5 y otra de 1, es decir 4 monedas en total. La técnica que se aplica aquí es tratar de dar la mayor cantidad de monedas de la mayor denominación, al resto aplicarle el mismo principio con la siguiente de denominación y así[7]. Se puede demostrar que para esta distribución de las denominaciones de monedas (en 1, 5, 10 y 25) esta técnica siempre minimiza el número de monedas utilizadas[8].

La técnica anterior es un ejemplo de lo que se conoce como *algoritmo goloso* (*greedy*). En cada momento se trata de aplicar la decisión que parece "la mejor" (tomar las monedas de mayor valor posible) sin tener en cuenta las consecuencias futuras. Esta filosofía de *tomar lo más que se pueda ahora* es lo que da el nombre de goloso al algoritmo[9]. Esta técnica tiene la ventaja de que corresponde con la intuición y por lo general los algoritmos son relativamente fáciles de programar (ejercicio 20). Sin embargo, no siempre producen el mejor resultado. Por ejemplo si además de las denominaciones anteriores se acuñara una nueva moneda de 20 centavos entonces el mejor modo de dar cambio para 41 centavos es con 2 monedas de 20 y una moneda de 1, es decir 3 monedas en total.

El problema general a resolver es encontrar un algoritmo para saber cómo dar el cambio para un conjunto cualquiera de denominaciones de monedas utilizando la menor cantidad de monedas posibles. Note la generalidad y aplicabilidad de este problema: en lugar de monedas y cambio se podría estar hablando, por ejemplo, de capacidades en litros de tipos de envases y el problema de cómo entregar un cierto volumen en litros utilizando la menor cantidad de envases posibles.

La programación dinámica en este caso ayudaría a evitar repetir cálculos para sub-problemas que tienen sub-subproblemas en común (Figura 9-13).

[6] Lo cual debería ser el caso de la cajera de un supermercado aunque no siempre ocurre así.
[7] Eso es muy simple y es lo que hace (o debiera hacer) automáticamente la cajera del supermercado.
[8] No en balde muchos países han adoptado esta distribución de denominaciones.
[9] ¿No le ha ocurrido alguna vez que por haber sido muy goloso durante la cena no le ha quedado "espacio" para al final disfrutar del postre?

Draft para estudiantes de

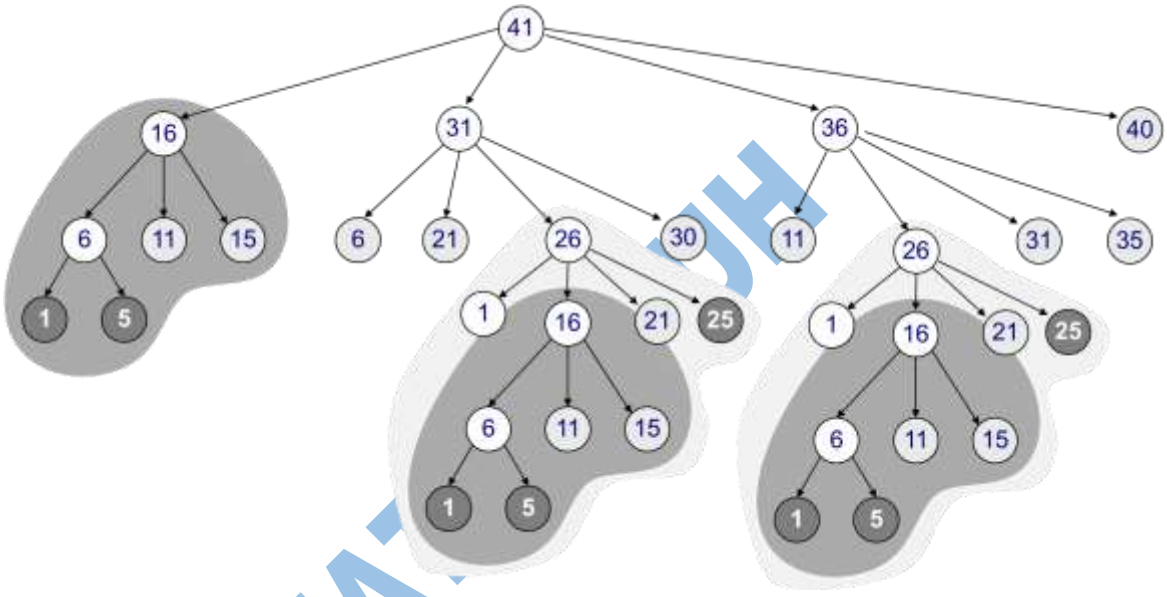


Figura 9-13 Cálculo redundante en diversos subproblemas para obtener 41 en monedas de 25, 10, 5 y 1 céntimos.

Como fórmula general para obtener una solución dinámica para un problema se pueden seguir los siguientes pasos:

1) Encontrar la subestructura óptima en los subproblemas.
2) Determinar la función recursiva que representa la dependencia entre los óptimos.
3) Calcular los valores de los óptimos para los subproblemas en sentido *bottom-up*.
4) Reconstruir una solución óptima del problema.

¿Cómo determinar la subestructura óptima en el caso del cambio en monedas?

Para ello debemos definir cuál es nuestro problema, e identificar cuáles son los posibles subproblemas y si existe una relación de optimalidad entre ellos.

En nuestro caso el problema está dado por el conjunto de las denominaciones ($\{c_i\}$) y un valor de cambio (T). Como para obtener T se "suman" monedas, un problema más simple del que puede depender el problema de dar cambio T es obtener cualquier cambio menor, es decir un cambio $t < T$ con el mismo conjunto de denominaciones (tenemos cantidad indefinida de monedas de cada tipo).

Una solución particular de nuestro problema deberá expresar qué monedas debemos tomar (secuencia $m_1, \dots, m_k$ donde cada $m_j \in \{c_i\}$) de forma que se obtenga el cambio deseado:

$$\sum m_j = T$$

Y que la cantidad de monedas (k) sea mínima.

Supongamos que tenemos un óptimo para el problema obtener un cambio T, sea m_k el último valor de moneda seleccionado... ¿existe relación entre el óptimo del problema de obtener un cambio $T - m_k$ y el de obtener un cambio T?

udiantes de MATCOM

En efecto, no se puede obtener un cambio para $T - m_k$ con menos de $k-1$ monedas y existe al menos uno con exactamente $k-1$ monedas $(m_1 + m_2 + \cdots + m_{k-1})$. La demostración de la primera afirmación se puede hacer por reducción al absurdo. Suponemos que existe tal cambio $T - m_k = a_1 + a_2 + \cdots + a_t$ con $a_j \in \{c_i\}$ y $t < k-1$. Pero de ser así, existiría el cambio para $T = a_1 + \cdots + a_t + m_k$ con $t+1 < k$ monedas lo que contradice que k sea la cantidad óptima de monedas para T.

Es decir, el óptimo para T se puede obtener a partir del óptimo para un $t < T$.

¿Cómo quedaría expresada la dependencia recursiva?

Para obtener el óptimo $f[T]$[10] basta con probar de todas las posibles "últimas monedas" cuál es la que produce el menor cambio para el subproblema subyacente. Es decir:

$$f[T] = \begin{cases} 0 & T = 0 \\ \infty & T < 0 \\ 1 + \min\limits_{m \in c_i} f[T-m] & e.o.c. \end{cases}$$

El primer caso representa el hecho de que para cambiar "0" céntimos se necesitan 0 monedas. El segundo caso representa que nuestra tabla evalúa "infinito" para casos negativos. Esto es conveniente para poder referirnos a $f[T-m]$ sin tener en cuenta si efectivamente, $m \leq T$. El último caso representa la dependencia entre los óptimos del problema T y alguno de los subproblemas $T-m$.

La ejecución *bottom-up* de nuestro algoritmo únicamente debe calcular los valores $f[0]$, $f[1]$, ..., $f[T]$ en ese orden. La implementación en C# puede verse en el . Note que estamos asignando valor infinito a los problemas que no tienen cambio posible, por ejemplo, cambios negativos.

```
public static int DevolverCambioDyn(int T, int[] c) {
    int[] mem = new int[T + 1];
    mem[0] = 0;
    for (int t = 1; t <= T; t++) {
        mem[t] = int.MaxValue;
        foreach (var m in c)
            if (t - m >= 0 && mem[t - m] != int.MaxValue)
                if (mem[t] > mem[t - m] + 1)
                    mem[t] = mem[t - m] + 1;
    }
    return mem[T];
}
```

Listado 9-11 Algoritmo con programación dinámica para dar solución al cambio en monedas

¿Cómo podemos reconstruir la solución particular para la que se obtuvo el óptimo?

Una primera solución puede ser incluir una seguna tabla que represente para cada subproblema t la opción con la que se obtuvo el óptimo. El listado __ muestra esta solución.

```
public static int[] DevolverSolCambio(int T, int[] c)
{
    int[] mem = new int[T + 1];
```

[10] Utilizamos una notación con corchetes ([]) para dejar claro que f es una tabla a ser rellenada en lugar de una función a ser ejecutada.

Draft para estudiantes de

```
    int[] opc = new int[T + 1];
    mem[0] = 0;
    for (int t = 1; t <= T; t++) {
        mem[t] = int.MaxValue;
        foreach (var m in c)
            if (t - m >= 0 && mem[t - m] != int.MaxValue)
                if (mem[t] > mem[t - m] + 1) {
                    mem[t] = mem[t - m] + 1;
                    opc[t] = m;
                }
    }
    int[] sol = new int[mem[T]];
    int N = T;
    int count = 0;
    while (N > 0) {
        sol[count++] = opc[N];
        N -= opc[N];
    }
    return sol;
}
```

Listado 9-12 Solución con programación dinámica al problema del cambio en monedas, reconstruyendo la solución óptima

Para reconstruir la solución basta comenzar con el valor inicial T y consultar la opción que se tomó para obtener el mínimo en ese subproblema (almacenado durante la programación dinámica en opc[T]). Luego, se resta ese mismo valor para continuar reconstruyendo la solución con las demás "opciones" que se tomaron.

Si se desea ahorrar memoria, se puede recuperar para cada valor de $f[T] = k$, ¿cuál es la m que cumple que $f[T - m] = k - 1$?

Repitiendo este proceso hasta llegar a 0 se puede recuperar la secuencia $m_k, m_{k-1}, \ldots, m_1$.

Dejamos al lector esta implementación. La Figura 9-14 muestra la ejecución del algoritmo para lograr un cambio de 12 con denominaciones $\{1, 5, 10, 25\}$.

Estudiantes de MATCOM UH

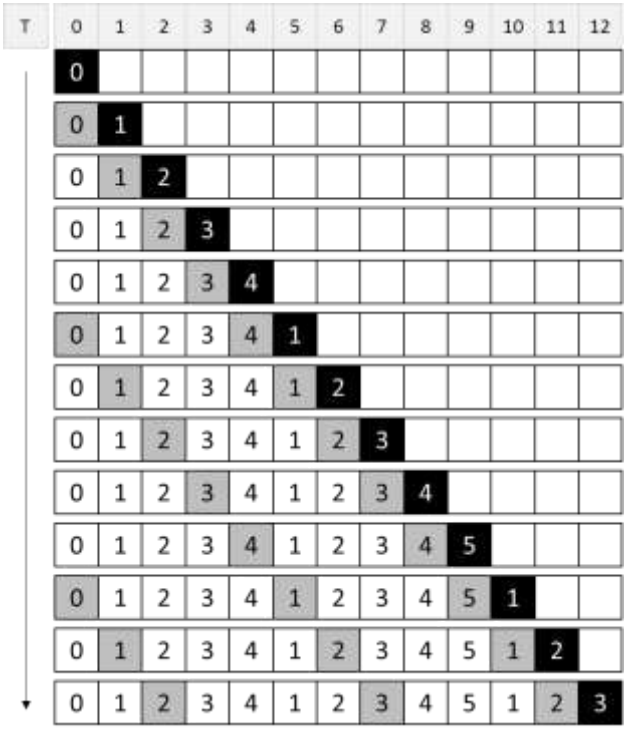


Figura 9-14 Estados de la tabla dinámica durante la ejecución del devolver cambio de 12 en denominaciones 1, 5, 10 y 25.

Cada fila muestra el valor de `t` en cada iteración externa. En gris, los valores válidos considerados para `t-m` y en negro el valor final (mínimo) calculado. Finalmente, para 12 se obtiene un valor óptimo de 3. ¿Cómo podríamos conocer qué 3 monedas necesitamos? Observando vemos que existen dos valores consultados que dan el mínimo (`f[12-1]` y `f[12-10]`). Ello corresponde al hecho que no importa el orden en que se tomen las monedas, el mínimo siempre es 3 (10+2 o 1+11). Si vamos a la fila 2 (porque nos quedamos con la opción `f[12-10]`, es decir, tomar una moneda de 10), encontramos el mínimo en `f[2-1]`, y luego en la fila 1 encontramos el mínimo en `f[1-1]`. La solución final reconstruida fue `10+1+1`.

A continuación veremos otro ejemplo, esta vez donde la "tabla" que se requiere es de dos dimensiones.

Subsecuencia palíndrome más larga

Recuerde la definición de que una cadena se considera un palíndromo cuando tiene los mismos caracteres, lo mismo mirando la cadena de izquierda a derecha que de derecha a izquierda. Por ejemplo la cadena `"anitalavalatina"` es un palíndromo.

El problema que se quiere resolver ahora es a partir de una cadena determinar cuál es la longitud de la subsecuencia más larga de esa cadena que forma un palíndromo. Por ejemplo la subsecuencia más larga de la cadena `"LOCOLOCOYLOQUITO"` que forma un palíndromo es la que se indica en cursiva `"LOCOLOCOYLOQUITO"` que formaría la cadena `"LOCOLOCOL"` la cual tiene longitud 9. Note que hablamos de subsecuencia y no estrictamente de subcadena. La subcadena más larga que forma un palíndromo sería `"OCOLOCO"` que tiene longitud 7.

Draft para estudiantes de

Para cada problema definido como una cadena S con n caracteres, se tienen 2^n posibles sub-problemas. No obstante, de todos ellos sólo es necesario considerar tres. El que resulta de eliminar el primer carácter (subcadena $S[1\ldots n-1]$), el que resulta de eliminar el último carácter (subcadena $S[0\ldots n-2]$) y el que resulta de eliminar a la vez el primero y el último ($S[1\ldots n-1]$). Cualquier otra combinación puede resultar como un sub-subproblema de éstos, a algún nivel.

Analicemos cómo queda reflejada la subestructura óptima en estos subproblemas.

1) Si el primer y último carácter de S son iguales, entonces existe una subsecuencia máxima palíndrome en S que los contiene.

Demostración:

Supongamos que no.

Si ninguno de los dos extremos pertenece a una subsecuencia óptima entonces se cumple que una solución óptima para S puede ser una subsecuencia palíndrome óptima en la subcadena $S[1\ldots n-1]$ de k caracteres. Pero a esta subsecuencia se le pueden añadir los caracteres $S[0]$ y $S[n-1]$ y seguiría siendo palíndrome con $k+1$ caracteres, lo que contradice la optimalidad supuesta para S.

Si sólo uno de los dos pertenece a una subsecuencia óptima, sin perder generalidad, supongamos que $S[0]$ está en la subsecuencia óptima y $S[n-1]$ no puede estar. Entonces existe un $S[t]$ que es el igual a $S[0]$ y es el último carácter de la subsecuencia óptima. Pero por transitividad, $S[t]=S[0]=S[n-1]$, se puede remplazar $S[t]$ por $S[n-1]$ obteniéndose una subsecuencia óptima de igual tamaño y contradiciendo la premisa que $S[n-1]$ no pueda estar en la subsecuencia óptima.

2) Si el primer y último carácter de S son distintos, entonces uno de los dos NO puede estar en la subsecuencia óptima (una subsecuencia palíndrome comienza y termina con el mismo carácter). Por tanto, la subsecuencia palíndrome más larga se encuentra o en la subcadena S[1...n-1] (suponiendo que S[0] sea el carácter que no puede estar en la solución óptima, o en la subcadena S[0...n-2] (suponiendo que S[n-1] sea el carácter evitado).

Estos dos resultados nos permiten determinar las dependencias de optimalidad entre los subproblemas.

$$P(S)=\begin{cases} 0 & |S|=0 \\ 1 & |S|=1 \\ 2+P(S[1\ldots n-2] & S[0]=S[n-1]\ (1) \\ \max\big(P(S[1\ldots n-1]),P(S[0\ldots n-2])\big) & S[0]\neq S[n-1]\ (2) \end{cases}$$

Esta formulación recursiva la podemos analizar en forma de tabla si consideramos siempre el mismo valor de S, y nos referirmos a un subproblema como dos índices, inicio y fin (i,j).

La forma de "llenar" la tabla quedaría en función de los valores i y j de la forma:

Estudiantes de MATCOM UH

$$p[i,j] = \begin{cases} 0 & i > j \\ 1 & i = j \\ 2 + p[i+1, j-1] & S[i] = S[j] \\ \max(p[i+1,j], p[i,j-1]) & S[i] \neq S[j] \end{cases}$$

Note que en la tabla, para valores de la fila i ¡se depende de valores de la fila $i+1$! ¿Quiere decir ello que dependemos de subproblemas más grandes? No, en realidad la "complejidad" (tamaño) de nuestro subproblema está dada por la expresión $j - i + 1$. Esto lo que significa es que tenemos que llenar la tabla en otro orden distinto al que estamos acostumbrados (incremento primero por fila y luego por columna).

Nuestro algoritmo deberá comenzar obteniendo los valores de los subproblemas de tamaño 1 (es decir $j - i + 1 = 1$). Estas serían las celdas con $i = j$ (diagonal). Luego los de la diagonal superior a esta ($j - i + 1 = 2$), serían las celdas con $i = j - 1$. El listado muestra la solución. La Figura 9-15 muestra la ejecución de este algoritmo y cómo finalmente la tabla.

```
public static int MayorSubsecuenciaPalindromo(string s) {
    int[,] tab = new int[s.Length, s.Length];
    for (int d = 0; d < s.Length; d++)
        tab[d, d] = 1;
    for (int k=2; k<=s.Length; k++) // tamaño del subproblema
        for (int i = 0; i <= s.Length - k; i++) { // posibles inicios
            int j = i + k - 1; // final
            if (s[i] == s[j])
                tab[i, j] = 2 + tab[i + 1, j - 1];
            else
                tab[i, j] = Math.Max(tab[i + 1, j], tab[i, j - 1]);
        }
    return tab[0, s.Length - 1];
}
```

Listado 9-13 Algoritmo con programación dinámica para determinar la mayor subsecuencia palíndromo

En esta tabla todos los valores de la triangular inferior quedan igualados a 0 por el primer caso ($i > j$). Los valores de la diagonal quedan iguales a 1 por el segundo caso ($i = j$). La tabla se llena por diagonales, comenzando por la diagonal principal, hacia arriba. En cada celda, las de color gris corresponden al tercer caso ($S[i] = S[j]$) y por tanto su valor es 2 unidades (los extremos) por encima del óptimo calculado para la posición $[i+1, j-1]$ (diagonal izquierda-abajo). En color blanco quedan representados los casos ($S[i] \neq S[j]$), donde el valor es el máximo entre la celda de la izquierda (posición $[i, j-1]$) y la celda de abajo (posición $[i+1, j]$). En la última posición de la primera fila quedaría el largo de la mayor subsecuencia palíndrome.

Draft para estudiantes de

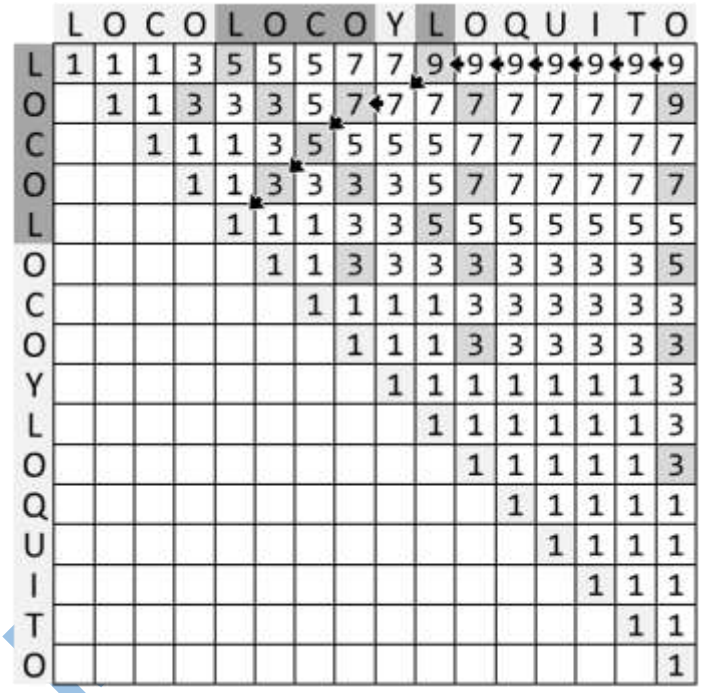


Figura 9-15 Tabla dinámica resultante para el problema "LOCOLOCOYLOQUITO"

¿Cómo podemos recuperar los caracteres que formaron parte de dicha solución óptima? Podemos seguir el rastro del algoritmo de atrás hacia delante, es decir, considerar ¿por qué apareció el 9 en la última casilla? Podemos para cada casilla determinar de dónde vino su valor, y aquellas casillas por las que pasemos que cumplan la condición $S[i] = S[j]$, determinarán los caracteres que forman parte de la subsecuencia. La Figura 9-15 muestra este recorrido (siempre que fueron iguales se escogió el valor de la izquierda).

Se invita al lector a que obtenga las soluciones dinámicas de los problemas vistos en el capítulo de recursividad a los que se pueda aplicar. Entre los ejemplos clásicos utilizados para ilustrar la programación dinámica se encuentran: la cadena de multiplicación de matrices y la subsecuencia común más larga.

9.7.2 Algoritmos golosos

En los problemas de optimización (como el problema de la mochila, el viajante y el cambio en monedas), existen decisiones que podemos tomar que representan una mejora local con respecto a otras, y por tanto parecería intuitivo optar por ellas en primera instancia. Por ejemplo, escoger colocar primero en la mochila el elemento que mayor valor obtenga en la razón ganancia sobre peso; que el viajante visite como próxima ciudad aquella que no esté visitada y le quede más cerca; intentar primero con las monedas de mayor denominación para devolver un cambio.

En todos estos casos estas estrategias golosas (*greedy* en inglés, traducido en ocasiones como voraces), intentan obtener mediante decisiones óptimas locales (el próximo

elemento a escoger, la próxima ciudad a visitar, la próxima moneda a utilizar), una solución óptima global. En la mayoría de los problemas de optimización esto no es viable.

Las estrategias golosas suelen representar aproximaciones a las soluciones óptimas pero no siempre collevan a una solución exacta del problema de optimización. Sin embargo; existen problemas para los que el enfoque ***greedy*** resuelve encontrar una solución óptima al problema.

En este epígrafe se verán algunos ejemplos y de qué forma se puede demostrar la factibilidad de la estrategia para obtener una solución óptima.

Cambio en monedas con denominaciones de la forma c^i

En el problema general del cambio en monedas no se puede utilizar una estrategia *greedy* para obtener la solución óptima (un contraejemplo se puede crear fácil tratando de obtener el cambio de 8 centavos con denominaciones de 1, 4 y 5), para el caso en que las denominaciones sean potencias de un valor $c \geq 2$ (por ejemplo, 1, 2, 4, 8, ...) la solución greedy funciona y permite obtener el óptimo global.

Cualquier cambio que se obtenga utiliza de cada denominación no más de $c - 1$ monedas. Si una solución óptima utiliza c monedas de la denominación c^i, se podría utilizar en su lugar una única moneda con denominación $c^{i+1} = c * c^i$.

Demostraremos entonces que dado un cambio T, en el óptimo siempre debe utilizarse la mayor denominación que cumpla $c^i \leq T$

Demostración:

Supongamos que no. Sea c^i el mayor valor que cumple que $c^i \leq T$, se tiene que el mayor valor posible a expresar con las denominaciones restantes es:

$$T' = (c-1)c^{i-1} + \cdots + (c-1)c^1 + (c-1)c^0 = (c-1)\frac{c^i-1}{c-1} = c^i - 1 < c^i.$$

Lo que contradice que T se pueda expresar con denominaciones menores que c^i.

En este problema se pudo demostrar de forma sencilla la factibilidad de la estrategia golosa porque existe sólo una solución óptima. A continuación veremos un ejemplo en que la demostración de la factibilidad de la estrategia debe lidiar con la existencia de varias soluciones óptimas.

Atravesando intervalos

Se tiene un conjunto de n intervalos cerrados $I = \{[a_i, b_i]\}$, se desea conocer el mínimo conjunto de puntos en $\mathbb{R}$ $\{x_i\}$, tal que todo intervalo en I contiene a algún punto de $\{x_i\}$.

Draft para estudiantes de

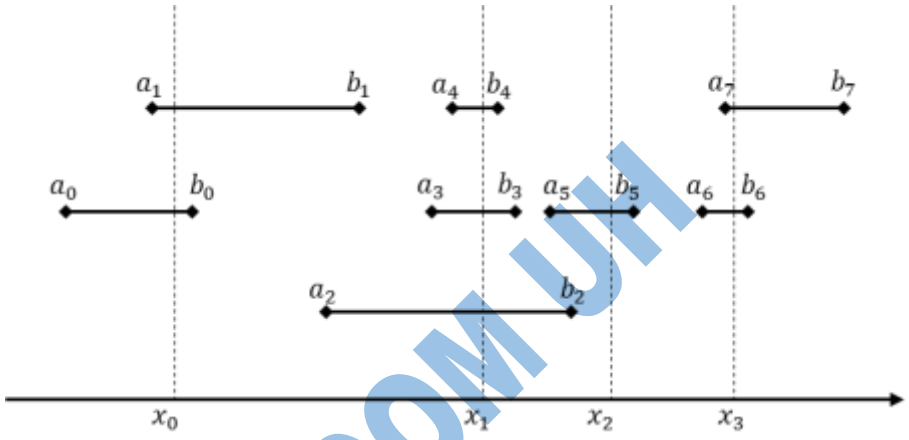


Figura 9-16 Problema de los segmentos atravesados

Supongamos que los intervalos están ordenados por el valor a_i y que el conjunto resultante se devuelve ordenado. La Figura 9-16 muestra una posible solución óptima para determinada entrada.

¿Cuál sería la estrategia golosa en este problema? Intuitivamente debemos tratar de utilizar la menor cantidad de puntos para cubrir la mayor cantidad de segmentos, por lo tanto, ¿dónde ubicar el primer punto? Lo ideal sería lo más a la derecha posible siempre que no rebase el valor del final más a la izquierda de los intervalos no cubiertos, es decir, en el menor b_i de los intervalos por cubrir.

Luego, se eliminan todos los intervalos que contienen a x_0 y se continúa el algoritmo con los restantes (Figura 9-17).

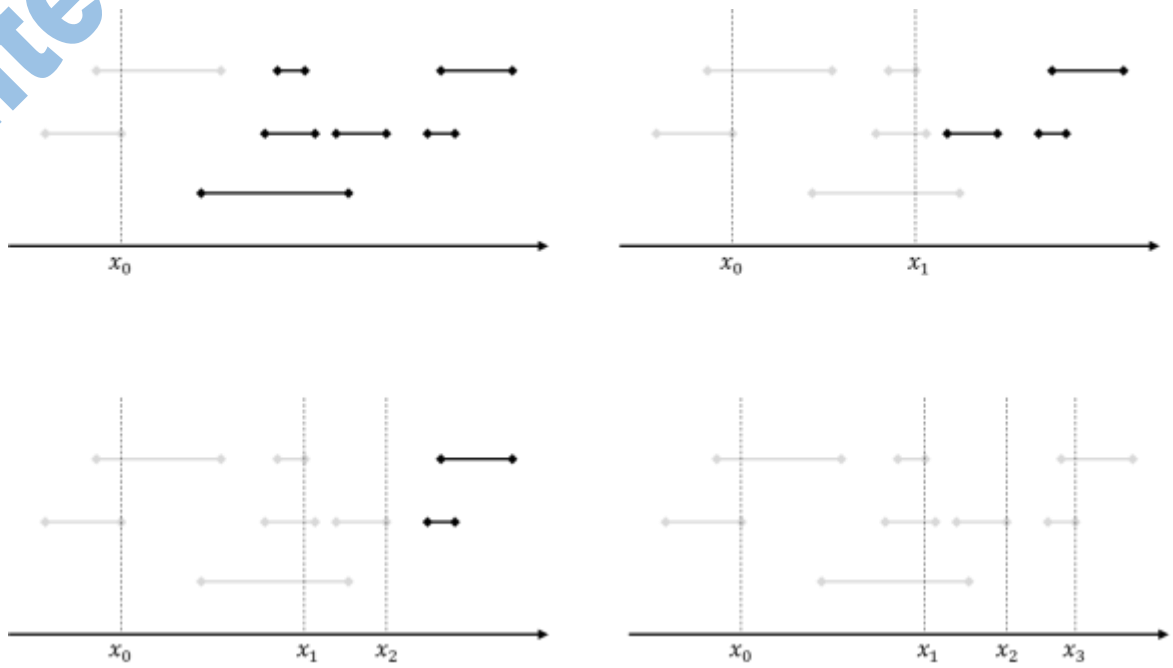


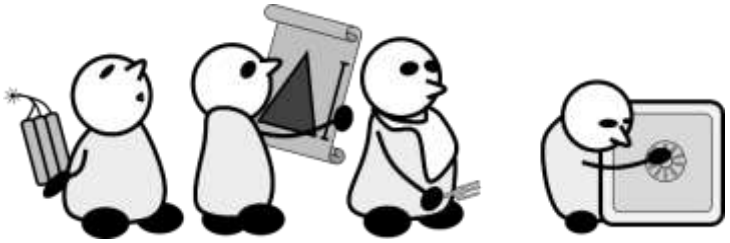
Figura 9-17 Decisiones tomadas en cada paso de nuestro algoritmo goloso.

Demostraremos a continuación que esta estrategia resuelve al menos una de las soluciones óptimas del problema.

Estudiantes de MATCOM UH

Supongamos que nuestra solución es el conjunto $\{ x_0, x_1, \dots, x_{k-1} \}$. De todas las posibles soluciones óptimas del problema $\{a_o, a_1, \dots, a_{t-1}\}$ tomemos la que tiene el mayor prefijo común con la nuestra. Si el mayor prefijo es la propia secuencia entonces nuestra solución está entre los óptimos. Si no, se tiene que $a_i = x_i$ para todos los $i < m$ y $a_m \neq x_m$.

Obligatoriamente se debe cumplir que $a_m < x_m$. De no ser así, el intervalo cuyo extremo derecho es x_m quedaría sin ser intersectado. En este paso del algoritmo, todos los intervalos que no han sido atravesados y que comienzan antes de a_m terminan después de x_m (puesto que nuestro algoritmo escoge el menor de todos los terminales restantes), por lo tanto, se puede cambiar el punto a_m por el x_m sin que empeore la solución óptima, lo que contradice el hecho de que $a_m \neq x_m$. Es decir, nuestra solución está siempre entre las óptimas al problema. Invitamos al lector a ofrecer una implementación de estos problemas.



9.8 El Mito P=NP

¿No existirá algún algoritmo que pueda resolver el problema del viajante en un costo polinomial? Desafortunadamente no se conoce aún. Encontrar dicha solución sería probablemente la noticia del milenio para nuestra ciencia, incluso encontrar una demostración que pruebe que dicha solución no existe. A esto se conoce como el problema de determinar si $P = NP$.

Un problema de decisión es aquel que obtiene una respuesta de Sí o No para determinada entrada. Por ejemplo: ¿es x primo? ¿Está el array a ordenado? ¿Existe una solución en el viajante con costo del trayecto menor que K?

El conjunto P representa la familia de estos problemas que tienen una solución con costo polinomial. El conjunto NP representa la familia de los problemas de decisión que son *solubles con costo polinomial en una máquina de Turing No-Determinista*. Si esto último no le dijo mucho, considere esta proposición equivalente: se puede comprobar determinada solución al problema en tiempo polinomial. Por ejemplo, saber si existe un camino del viajante que sea menor que un K nos obligaría a recorrer todos los posibles trayectos ($n!$) y hasta el momento no se conoce un algoritmo polinomial capaz de hacerlo. Sin embargo; para determinar si la solución específica $\{ c_1, c_2, \dots, c_n\}$ cuesta menos que K basta con realizar un recorrido lineal y sumar todos los caminos directos de la ciudad c_i a la c_{i+1}.

Intuitivamente puede verse que P está incluido en NP. La pregunta sin respuesta aún es si NP es también subconjunto de P (lo que haría iguales a los conjuntos).

Draft para estudiantes de

En 1971 Stephen Cook[11] propuso una familia de problemas conocida posteriormente como NP-Completos. Lo interesante de esta familia es que para cualquier par de problemas A y B, existe un algoritmo polinomial que puede transformar cualquier entrada de A en una entrada de B, y obtenerse mediante la respuesta de B en un tiempo polinomial la respuesta correspondiente al problema A. Este tipo de reducción de un problema en otro es una herramienta muy poderosa para el análisis de algoritmos, y en este caso, Cook la propuso para obtener una familia de problemas NP que son tan "polinomiales" uno como otro. En otras palabras, si se encuentra una solución polinomial para uno de los problemas NP-Completo, se estaría encontrando una solución polinomial ¡para todos! y si se demuestra una cota inferior asintótica exponencial para uno, se estaría demostrando que $P \neq NP$.

Desde entonces se han incorporado un sinfín de problemas a la familia NP-completos, entre los que destacan los problemas de decisión del viajante y la mochila.

Si un problema no es de decisión (por ejemplo, determinar el menor camino en el viajante), pero existe un problema NP que puede reducirse a éste, se dice entonces que el problema es NP-Difícil (*NP-Hard*), en otras palabras, tan "difícil" como un NP.

Ejemplificando nuevamente con el problema del viajante, el problema de decisión del viajante es "*Existe un recorrido con costo menor o igual que* K (con respuesta de Sí o No)". Si se obtiene el menor camino (problema que no es de decisión) y este costo es menor o igual que K, la respuesta al primer problema sería Sí, si no, sería No. Es decir, obtener el menor camino obligatoriamente es al menos, "tan difícil" como determinar si existe un trayecto con costo menor que un determinado valor K.

La teoría de la computabilidad y el análisis de algoritmos es un tema que escapa al alcance de este libro, sin embargo, una pequeña introducción a estos temas podría atrapar la atención de un lector que... quién sabe, podría ser quien finalmente desentrañe el mito $P = NP$.[12]

[11] Cook fue premiado con el Premio Turing en ... por sus resultados en teoría de la complejidad de los algoritmos.
[12] Este problema representa un desafío tan relevante para nuestra ciencia que el instituto .... tiene un premio de un millón de dólares al primero que obtenga una demostración formal de $P = NP$ o $P \neq NP$.

udiantes de MATCOM UH