dos como con *ordenar por fusión*. La demostración de esta conjetura es una bonita aplicación de la técnica de inducción constructiva (Sección 1.6.4). Para aplicar esta técnica, postulamos la existencia de una constante c, desconocida por el momento, tal que $t(n) \le c\, n \log n$ para todo $n \ge 2$. Hallaremos un valor adecuado para esta constante en el proceso de demostración de su existencia por inducción matemática generalizada. Comenzamos con $n = 2$, porque $n \log n$ está indefinido o es nulo para valores más pequeños de n; alternativamente, podríamos empezar en $n = n_0 + 1$.

Teorema 7.4.1 *Quicksort requiere un tiempo en $O(n \log n)$ para ordenar n elementos en el caso medio.*

Demostración. Sea $t(n)$ el tiempo requerido por *quicksort* para ordenar n elementos en el caso promedio. Sean d y n_0 constantes tales que es válida la ecuación 7.2. Deseamos demostrar que $t(n) \le c\, n \log n$ para todo $n \ge 2$, siempre y cuando c sea una constante elegida adecuadamente. Procedemos por inducción constructiva. Supongamos sin pérdida de generalidad que $n_0 \ge 2$.

◊ *Base*: consideremos cualquier entero n tal que $2 \le n \le n_0$. Tenemos que demostrar que $t(n) \le c\, n \log n$. Esto es fácil, puesto que tenemos total libertad para seleccionar la constante c, y el número de casos es finito. Basta con seleccionar c que sea al menos tan grande como $t(n)/(n \log n)$. De esta manera, nuestra primera restricción sobre c es:

$$c \ge \frac{t(n)}{n \log n} \text{ para todo } n \text{ tal que } 2 \le n \le n_0 \tag{7.3}$$

◊ *Paso de inducción*: consideremos cualquier entero $n > n_0$. Se toma como hipótesis de inducción que $t(k) \le c\, k \log k$ para todo k tal que $2 \le k < n$. Deseamos restringir c de tal manera que $t(n) \le c\, n \log n$ se siga de la hipótesis de inducción. Supongamos que a representa $t(0) + t(1)$. Comenzando con la ecuación 7.2:

$$\begin{aligned} t(n) &\le dn + \frac{2}{n}\sum_{k=0}^{n-1} t(k) \\ &= dn + \frac{2}{n}\left(t(0) + t(1) + \sum_{k=2}^{n-1} t(k)\right) \\ &\le dn + \frac{2a}{n} + \frac{2}{n}\sum_{k=2}^{n-1} ck \log k \text{ por la hipótesis de inducción} \\ &\le dn + \frac{2a}{n} + \frac{2c}{n}\int_2^n x \log x\, dx \text{ (véase la figura 7.4)} \end{aligned}$$

$$\begin{aligned} &= dn + \frac{2a}{n} + \frac{2c}{n}\left[\frac{x^2 \log x}{2} - \frac{x^2}{4}\right]_{x=2}^{n} \\ &\text{(recordemos que «log» representa el logaritmo natural)} \\ &< dn + \frac{2a}{n} + \frac{2c}{n}\left(\frac{n^2 \log n}{2} - \frac{n^2}{4}\right) \\ &= dn + \frac{2a}{n} cn \log n - \frac{cn}{2} \\ &= cn \log n - \left(\frac{c}{2} - d - \frac{2a}{n^2}\right) n. \end{aligned}$$

Se sigue que $t(n) \le c\, n \log n$ siempre y cuando $c/2 - d - 2a/n^2$ sea no negativo, lo cual equivale a decir que $c \ge 2d + 4a/n^2$. Dado que aquí consideramos solamente el caso en el cual $n > n_0$, todo va bien siempre y cuando

$$c \ge 2d + \frac{4a}{(n_0 + 1)^2} \tag{7.4}$$

que es nuestra segunda y última restricción sobre c.
Reuniendo las restricciones dadas por las ecuaciones 7.3 y 7.4, basta hacer

$$c = \text{máx}\left(2d + \frac{4(t(0) + t(1))}{(n_0 + 1)^2}, \text{máx}\left\{\frac{t(n)}{n \log n} \middle| 2 \le n \le n_0\right\}\right) \tag{7.5}$$

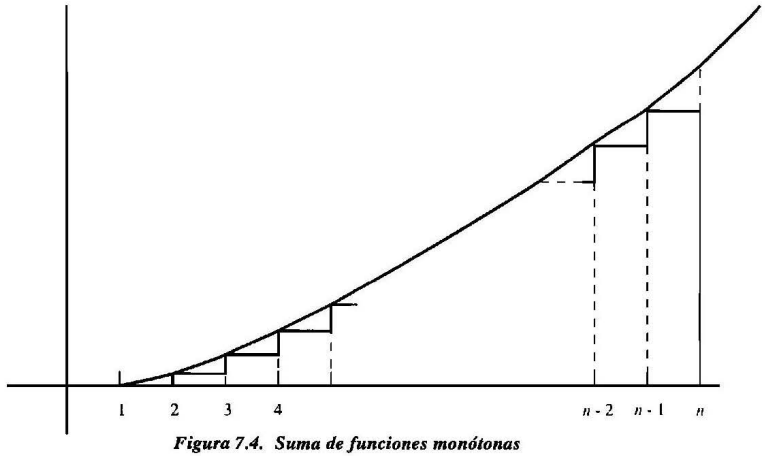

Figura 7.4. Suma de funciones monótonas

para finalizar la demostración por inducción constructiva de que $t(n) \leq c\, n \log n$ para todo $n \geq 2$, y por tanto que $t(n) \in O(n \log n)$.
Si se siente confuso o poco convencido, le animamos a que desarrolle por sí mismo una demostración por inducción matemática generalizada —en vez de inducción matemática constructiva— de que $t(n) \leq c\, n \log n$ para todo $n \geq 2$, utilizando esta vez el valor explícito de c dado por la ecuación 7.5.

Por tanto, *quicksort* puede ordenar una matriz de n elementos distintos en un tiempo promedio que está en $O(n \log n)$. En la práctica, la constante oculta es más pequeña que las asociadas en *ordenación por montículo* o en *ordenar por fusión*. Si ocasionalmente se puede tolerar algún tiempo de ejecución largo, éste es un excelente algoritmo de propósito general. ¿Se puede modificar *quicksort* para que requiera un tiempo que esté en $O(n \log n)$ incluso en el caso peor? La respuesta es sí pero no. Incluso si seleccionamos la mediana de $T[i..j]$ como pivote, lo cual se puede hacer en un tiempo lineal según se verá en la Sección 7.5, *quicksort* tal como se ha descrito aquí sigue requiriendo un tiempo cuadrático en el caso peor, lo cual sucede si son iguales todos los elementos que hay que ordenar. Basta una sencilla modificación del algoritmo de partición para evitar este comportamiento incorrecto, aunque es complicado programarlo de forma correcta y eficiente. Para esto, se necesita un nuevo

procedimiento *pivotebis*($T[i..j]$, p; **var** k, l)

que descomponga T en *tres* secciones, empleando p como pivote: después de la partición, los elementos de $T[i..k]$ son más pequeños que p, los de $T[k+1..l-1]$ son iguales a p, y los de $T[l..j]$ son más grandes que p. Los valores de k y l son devueltos por *pivotebis*. Después de hacer la partición con una llamada a *pivotebis*($T[i..j]$, $T[i]$, k, l) hay que llamar a *quicksort* recursivamente, con $T[i..k]$ y $T[l..j]$. Con esta modificación, la ordenación de una matriz de elementos iguales requiere un tiempo lineal. Hay algo más interesante: ahora *quicksort* requiere un tiempo que está en $O(n \log n)$ incluso en el caso peor si se selecciona como pivote la mediana de $T[i..j]$ en un tiempo lineal. Sin embargo, sólo mencionamos esta posibilidad para indicar que hay que evitarla: la constante oculta asociada a esta versión «mejorada» de *quicksort* es tan grande que da lugar a un algoritmo, peor que *ordenación por montículo* en todos los casos.

7.5 BÚSQUEDA DE LA MEDIANA

Sea $T[1..n]$ una matriz de enteros, y sea s un entero entre 1 y n. Se define el *s-ésimo menor elemento* de T como aquel elemento que se encontraría en la s-ésima posición si se ordenara T en orden no decreciente. Dados T y s, el problema de encontrar el s-ésimo elemento de T se conoce como el *problema de selección*. En particular, se define la *mediana* de T como su $\lceil n/2 \rceil$-ésimo elemento. Cuando n es

impar y los elementos de T son diferentes, la mediana es simplemente aquel elemento de T tal que en T hay tantos elementos más pequeños que él como elementos mayores que él. Por ejemplo, la mediana de [3, 1, 4, 1, 5, 9, 2, 6, 5] es 4, puesto que 3, 1, 1 y 2 son más pequeños que 4, mientras que 5, 9, 6 y 5 son mayores.

¿Qué podría ser más sencillo que buscar el menor elemento de T, o calcular la media de todos los elementos? Sin embargo, no es evidente que sea posible calcular la mediana con tanta facilidad. Un algoritmo sencillo para determinar la mediana de $T[1..n]$ consiste en ordenar la matriz y extraer entonces el elemento $\lceil n/2 \rceil$*–ésimo*. Si utilizamos *ordenación por montículo* u *ordenación por fusión*, esto requiere un tiempo que está en $\Theta(n \log n)$. ¿Podemos hacerlo mejor? Para responder a esta pregunta, estudiaremos la interrelación entre hallar la mediana y seleccionar el s-ésimo elemento más pequeño.

Es evidente que todo algoritmo para el problema de selección se puede utilizar para hallar la mediana: basta con seleccionar el $\lceil n/2 \rceil$-ésimo elemento más pequeño. Curiosamente, la inversa también es cierta. Supongamos por el momento que está disponible un algoritmo *mediano*($T[1..n]$) que devuelve la mediana de T. Dada una matriz T y un entero s, ¿cómo se podría utilizar este algoritmo para determinar el s-ésimo elemento más pequeño de T? Sea p la mediana de T. Ahora usamos p como pivote, de forma muy parecida a *quicksort*, pero usando el algoritmo *pivotebis* presentado al final de la sección anterior. Recuerde que una llamada a *pivotebis*($T[1..n]$, p; **var** k, l) particiona a $T[i..j]$ en tres secciones: T se reorganiza de tal manera que los elementos de $T[i..k]$ sean más pequeños que p, los de $T[k+1..l-1]$ sean iguales a p, y los de $T[l..j]$ sean mayores que p. Tras una llamada a *pivotebis*(T, p, k, l), hemos terminado si $k < s < l$, puesto que entonces el s-ésimo elemento más pequeño de T es igual a p. Si $s \leq k$, entonces el s-ésimo elemento más pequeño de T es ahora el s-ésimo elemento más pequeño de $T[1..k]$. Por último, si $s \geq l$, entonces el s-ésimo elemento más pequeño de T es ahora el $(s - l + 1)$-ésimo elemento más pequeño de $T[l..n]$. En cualquier caso, hemos avanzado, porque o bien hemos terminado, o bien la submatriz que hay que considerar contiene menos de la mitad de los elementos, por cuanto p es la mediana de la matriz original.

Hay una gran similitud entre este enfoque y la búsqueda binaria (Sección 7.3), y de hecho el algoritmo resultante se puede programar de forma iterativa en lugar de hacerlo recursivamente. La idea clave consiste en emplear dos variables i y j, inicializadas a 1 y n respectivamente, y asegurarnos que en todo momento $i \leq s \leq j$, y que los elementos de $T[1..i-1]$ sean más pequeños que los de $T[i..j]$, que a su vez son más pequeños que los de $T[j+1..n]$. La consecuencia inmediata es que el elemento deseado se encuentra en $T[i..j]$. Cuando todos los elementos de $T[i..j]$ sean iguales, hemos acabado.

La figura 7.5 ilustra el proceso. Por sencillez, la ilustración supone que *pivotebis* se ha implementado de una manera que es intuitivamente sencilla, aun cuando una implementación realmente eficiente haría las cosas de otra manera.