Diseño de algoritmos

Jesús Bermúdez de Andrés. UPV-EHU
Guías para la solución de ejercicios: Divide y vencerás

Curso 2008-09

1. Para resolver cierto problema se dispone de un algoritmo trivial cuyo tiempo de ejecución $t(n)$ es cuadrático (i.e. $t(n) \in O(n^2)$). Se ha encontrado una estrategia *Divide y Vencerás* para resolver el mismo problema; dicha estrategia realiza $D(n) = n \lg n$ operaciones para dividir el problema en dos subproblemas de tamaño mitad que el original y $C(n) = n \lg n$ operaciones para componer una solución del original con la solución de dichos subproblemas. ¿Es la estrategia *Divide y Vencerás* más eficiente que la empleada en el algoritmo trivial?

   Solución:
   La función de coste del algoritmo divide y vencerás indicado, responde a la siguiente ecuación:

$$f(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ 2f(\frac{n}{2}) + 2n \lg n & \text{si } n > 1 \end{cases}$$

   Resolviendo por expansión, llegamos a la ecuación patrón

$$\begin{aligned} f(n) &= 2^i f(\frac{n}{2^i}) + 2n \sum_{j=0}^{i-1} \lg \frac{n}{2^j} \\ &\quad \text{que llevándola al caso básico, cuando } n = 2^i \\ &= nf(1) + 2n \log n \sum_{j=0}^{i-1} 1 - 2n \sum_{j=0}^{i-1} \log 2^j \\ &= cn + n(\lg n)^2 - n \lg n \\ &= \Theta(n(\lg n)^2) \end{aligned}$$

   Puesto que

$$\lim_{n \to \infty} \frac{n(\lg n)^2}{n^2} = 0$$

   ocurre que $n(\lg n)^2 \in o(n^2)$, y concluimos que el algoritmo divide y vencerás es más eficiente que el trivial.

2. Escriba un algoritmo para ordenar una lista enlazada de elementos utilizando la estrategia de ORDEN_POR_MEZCLA. Analice su algoritmo en tiempo y espacio y compárelo con la versión clásica utilizada para ordenar vectores.

   Solución:
   La estrategia de ORDEN_POR_MEZCLA consiste en dividir por la mitad el conjunto de elementos a ordenar, ordenar recursivamente cada uno de ellos, y mezclar ordenadamente cada mitad.

   En el caso de que el conjunto de elementos a ordenar esté representado mediante una lista enlazada, con acceso secuencial a cada elemento (partiendo del primero) como único modo de alcanzar cada elemento, la tarea de dividir por la mitad el conjunto de elementos consiste en un recorrido de la lista hasta llegar al elemento que ocupe la posición $\lfloor \frac{n}{2} \rfloor$. Esto resulta de $\Theta(n)$.

   La mezcla ordenada de dos listas ordenadas se realiza de manera análoga a como se hace con vectores; pero con la ventaja de que no es necesario espacio extra de $\Theta(n)$, basta con recolocar los nodos de las listas mediante redireccionamiento de las referencias. Este procedimiento necesita un tiempo de $\Theta(n)$.

   En consecuencia, la función de coste temporal de esta versión de ORDEN_POR_MEZCLA es

   $$f(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ 2f(\frac{n}{2}) + \Theta(\text{n}) & \text{si } n > 1 \end{cases}$$

   que es la misma que la de la versión que ordena vectores. La diferencia de esta versión es que se reduce el espacio extra necesario; ahora el espacio extra proviene de la pila de llamadas recursivas, que alcanza un tamaño de $\Theta(\lg n)$.

3. Este ejercicio propone estudiar algunas variantes del algoritmo de ordenación por mezcla.

   a) Supóngase que queremos utilizar el algoritmo de ordenación por mezcla, pero dividiendo el vector en tres tercios, ordenando cada trozo recursivamente y mezclándolos después de ordenados. Escriba y analice esa versión del algoritmo, que denominaremos ORD_MEZCLA_3.

   b) Generalice el análisis para el caso de dividir el vector en $k \geq 3$ porciones de tamaño similar, ordenar recursivamente y mezclar las k porciones ordenadas.

   c) ¿Qué conclusiones puede extraer de lo anterior?

   d) Considere un algoritmo ORD_MEZCLA_RAÍZ que divida el vector en $\lfloor \sqrt{n} \rfloor$ trozos de tamaño $\lfloor \sqrt{n} \rfloor$ aproximadamente. Estudie con cuidado cómo debería ser el algoritmo de mezcla de los $\lfloor \sqrt{n} \rfloor$ trozos ordenados. Analice el algoritmo ORD_MEZCLA_RAÍZ y compárelo con los anteriores.

   e) Llevando al extremo la idea de dividir el vector cada vez en más porciones, para aplicar la idea de ordenación por mezcla. Considere un algoritmo que divida el vector en $n/2$ trozos de tamaño 2.

   Diseñe y analice un algoritmo siguiendo esta variante.

Solución:

a) El algoritmo ORD_MEZCLA_3 es totalmente análogo a la versión clásica de ordenación por mezcla de dos trozos. Simplemente hay que tener en cuenta que, en la rutina de mezcla de los tres trozos ordenados, hay que seleccionar el mínimo valor de tres valores, en vez de el mínimo de dos.

Por tanto el análisis de coste temporal termina con la resolución de una recurrencia del estilo siguiente:

$$f(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ 3f(\frac{n}{3}) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Con cualquiera de los métodos conocidos, se ha de llegar a comprobar que $f(n) = \Theta(n \lg n)$

b) Generalizar para el caso de $k \geq 3$ trozos, nos lleva a la recurrencia:

$$g(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ kg(\frac{n}{k}) + \Theta(n) & \text{si } n > 1 \end{cases}$$

Y se debe llegar a comprobar que $g(n) = \Theta(n \lg n)$

c) La conclusión de lo anterior es que la decisión de trocear el vector de entrada, en un número constante de porciones de tamaño similar (independiente del tamaño de la entrada n), no modifica la complejidad asintótica del algoritmo resultante.

d) La implementación de la idea de dividir el vector de entrada en $\lfloor\sqrt{n}\rfloor$ trozos de tamaño $\lfloor\sqrt{n}\rfloor$ aproximadamente, obliga a tener un poco más de cuidado con los detalles; pero no ha de ser muy distinta de la versión anterior para un número constante de trozos. El caso es que, ahora, el número de trozos depende del tamaño de la entrada. Si para mezclar los $\lfloor\sqrt{n}\rfloor$ trozos ordenados se ha seguido la misma pauta que en los apartados anteriores; es decir, calcular n veces el mínimo valor de un conjunto de (aproximadamente) $\lfloor\sqrt{n}\rfloor$ elementos, con el algoritmo trivial de recorrido de los elementos del conjunto, entonces la recurrencia a resolver debe ser del estilo:

$$t(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ \sqrt{n}\, t(\sqrt{n}) + n\sqrt{n} & \text{si } n > 1 \end{cases}$$

Si resolvemos esta ecuación por expansión, llegaremos a la expresión patrón $t(n) = n^{\frac{2^i-1}{2^i}} t(n^{\frac{1}{2^i}}) + n \sum_{j=1}^{i} n^{\frac{1}{2^j}}$. Y suponiendo que n es de la forma a^{2^k} tendremos $t(n) = \frac{n}{a}c + n \sum_{j=1}^{k} n^{\frac{1}{2^j}}$.

Obsérvese que $\sum_{j=1}^{k} n^{\frac{1}{2^j}} > n^{\frac{1}{2}} = \sqrt{n}$ y por lo tanto $t(n) = \Omega(n\sqrt{n})$ y concluimos que es una solución peor que la de los apartados anteriores.

El caso es que la rutina de la mezcla puede hacerse de un modo más eficiente. Usando una estructura de montículo (que se estudia en otro capítulo), se puede realizar la mezcla en tiempo $O(n \lg n)$.

e) En este caso, si se realiza la mezcla de los $n/2$ trozos ordenados de tamaño 2 calculando n veces el mínimo valor de un conjunto de (aproximadamente) $n/2$ elementos, con el algoritmo trivial de recorrido de los elementos del conjunto, entonces la recurrencia a resolver debe ser del estilo:

$$t(n) = \begin{cases} O(1) & \text{si } n \leq 2 \\ \Theta(n)\ t(2) + \Theta(n^2) & \text{si } n > 2 \end{cases}$$

Hay otras formas más eficientes de realizar la mezcla ordenada de los $n/2$ trozos ordenados de tamaño 2. Por ejemplo, usando una estructura de montículo (que se estudia en otro capítulo), la mezcla puede realizarse en tiempo $O(n \lg n)$.

4. Sea $T[1 \ldots n]$ un vector de enteros distintos ordenado de menor a mayor, algunos de los cuales pueden ser negativos. Diseñe un algoritmo que determine un valor $i \in \{1 \ldots n\}$ tal que $T[i] = i$, siempre que dicho i exista. Debe emplearse un tiempo $o(n)$.

   Solución:
   Puede utilizarse una estrategia similar a la de la *búsqueda dicotómica*: Preguntar por el valor de la componente que está en la mitad del vector que estamos estudiando, digamos que ese índice es m. Si $T(m) = m$ hemos terminado. Si $T(m) > m$ es imposible que el elemento que buscamos se encuentre a la derecha de m (¿por qué?) y deberíamos buscarlo en la parte izquierda. De manera análoga, si $T(m) < m$ es imposible que el elemento que buscamos se encuentre a la izquierda de m y deberíamos buscarlo en la parte derecha. Al codificar este algoritmo hay que tener cuidado con los detalles (igual que le ocurre al de *búsqueda dicotómica*). Es fácil analizar el algoritmo y descubrir que el tiempo será $O(\lg n)$.

5. Sea $T[1..n]$ un vector de números naturales distintos. Una pareja de valores $(T[i], T[j])$ es una *inversión* si no respetan el orden, es decir si $i < j$ pero $T[i] > T[j]$.

   Escriba un algoritmo de $O(n \lg n)$ que calcule el número de inversiones de un vector $T[1..n]$.

   Solución:
   Obsérvese que al ordenar un vector, deshacemos sus inversiones. Así que una solución consiste en adaptar el procedimiento ORDEN_POR_MEZCLA. La adaptación consistirá en añadir una variable $NInv$ que registre el número de inversiones que van deshaciéndose al ordenar.

```
func CUENTA_INVERSIONES (T: Tabla; Inicio, Fin: Natural) return natural
  NInv ← 0
  if Inicio < Fin then
    NInv ← CUENTA_INVERSIONES(T, Inicio, (Inicio + Fin)/2)
    NInv ← NInv + CUENTA_INVERSIONES(T, ((Inicio + Fin)/2) + 1, Fin)
    NInv ← NInv + MEZCLA_Y_CUENTA (T, Inicio, (Inicio + Fin)/2, Fin)
  end if
```

```
  return NInv

func Mezcla_Y_Cuenta (T: Tabla; I, Centro, F: Natural) return natural
  NInv ← 0
  Izq ← I
  Der ← Centro + 1
  A ← I     {primer subíndice de La_Mezcla}
  while Izq ≤ Centro ∧ Der ≤ F loop
    if T(Izq) ≤ T(Der) then
      La_Mezcla(A) ← T(Izq)
      Izq ← Izq+1
    else
      La_Mezcla(A) ← T(Der)
      Der ← Der+1
      NInv ← NInv + (Centro - Izq + 1)
    end if
    A ← A+1
  end loop
  if Izq > Centro then
    La_Mezcla(A..F) ← T(Der..F)
  else
    La_Mezcla(A..F) ← T(Izq..Centro)
  end if
  T(I..F) ← La_Mezcla(I..F)
  return NInv
```

Obsérvese que las sentencias que se incluyen para la adaptación de Orden_Por_Mezcla no modifican su orden de complejidad. Por lo tanto, Cuenta_Inversiones es de $O(n \lg n)$.

6. Sea $F(x)$ una función monótona decreciente y sea n el mayor entero que cumple $F(n) \geq 0$. Asumiendo que n existe, un algoritmo para determinar dicho n es el siguiente:

```
x ← 0
while F(x) ≥ 0 loop
  x ← x + 1
end loop
return x − 1
```

Compruébese que esta solución tiene complejidad $O(n)$. Escríbase un algoritmo cuyo comportamiento asintótico sea mejor en función de n.

Solución:
Puesto que se pide hallar una solución de orden mejor que lineal en n, esto indica que la búsqueda del número n no se debe realizar por incrementos

constantes (1, 2 o, en general, c constante), que siempre llevarían a un orden lineal, como puede comprobarse. En cambio, podemos buscar n realizando saltos que sean potencias de 2, hasta que hallemos un 2^k tal que $F(2^k) = 0$ (entonces, ese es el n que buscamos), o bien $F(2^k) < 0$, y eso significa que el n que buscamos está en el intervalo $(2^{k-1}, 2^k)$. En ese caso, buscamos dicotómicamente en ese intervalo de números. Al realizar saltos potencias de 2 conocemos con exactitud el número de elementos que hay en el intervalo en el que se realiza la búsqueda dicotómica, lo cual es imprescindible para determinar el orden de la misma.

```
func BuscaMaxPos(F) return natural
  k ← 0
  while F(2^k) > 0 loop
    k ← k + 1
  end loop
  if F(2^k) = 0 then return 2^k
  else return BusqDicotomica(F, 2^{k-1} + 1, 2^k − 1)
```

Sólo queda adaptar el algoritmo de búsqueda dicotómica a las circunstancias concretas de este problema.

El algoritmo BuscaMaxPos(F) tiene el siguiente coste:

a) Si $n = 2^k$ entonces se realizan k iteraciones del **while** (cada una de $O(1)$). Por lo tanto el algoritmo es $\Theta(\log n)$.

b) Si $2^{k-1} < n < 2^k$ entonces se realizan k iteraciones del **while** (cada una de $O(1)$), más las operaciones realizadas por BusqDicotomica(F, $2^{k-1} + 1$, $2^k - 1$). Como en el intervalo $(2^{k-1}, 2^k)$ hay 2^{k-1} elementos, se realizarán $\Theta(\log 2^{k-1}) = \Theta(k)$ preguntas en la búsqueda dicotómica. Por lo tanto el algoritmo será $\Theta(\log n)$ porque $k - 1 < \log n < k$.

Concluimos que BuscaMaxPos(F) es $\Theta(\log n)$ siendo n el mayor número entero tal que $F(n) \geq 0$.

7. Considere el siguiente algoritmo de ordenación:

```
proc OrdenSolapa(V, i, j)
  k ← ⌊(j−i+1)/3⌋
  if V[i] > V[j] then Intercambiar(V[i], V[j]) end if
  if j − i + 1 > 2 then
    OrdenSolapa(V, i, j − k)
    OrdenSolapa(V, i + k, j)
    OrdenSolapa(V, i, j − k)
```

- Determine el tiempo de ejecución y el espacio extra empleado por OrdenSolapa (en el peor caso) para ordenar un vector de tamaño n.

- Compare el tiempo de ejecución de OrdenSolapa con el tiempo de ejecución de los algoritmos de ordenación Orden_Por_Inserción, Orden_Por_Mezcla y Orden_Por_Montículo, para un vector de tamaño n.

(Nota: Para analizar el tiempo de ejecución de OrdenSolapa, considere únicamente valores de n de la forma $(\frac{3}{2})^k$. Obsérvese que $\lg_3 2 = 0{,}63092$.)

Solución:
En el algoritmo OrdenSolapa($V,\ i,\ j$), el valor $j-i+1$ representa el tamaño de la entrada. Entonces el tiempo de ejecución de OrdenSolapa($V,\ i,\ j$), con un tamaño de entrada n, queda especificado por la recurrencia:

$$t(n) = \begin{cases} a & \text{si } n \leq 2 \\ 3t(\frac{2n}{3}) + b & \text{si } n > 2 \end{cases}$$

Expandiendo la recurrencia, encontramos la ecuación patrón siguiente:

$$t(n) = 3^i\ t((\frac{2}{3})^i n) + b\sum_{j=0}^{i-1} 3^j$$

Suponiendo que $n = (\frac{3}{2})^k$ y que el caso básico se obtiene cuando $i = k$, tenemos

$$t(n) = 3^k a + b\ \frac{3^k - 1}{2} = (a + \frac{b}{2})3^k - \frac{b}{2}$$

Obtenemos el valor de k, en función de n, aplicando logaritmos a la ecuación $n = \frac{3}{2}^k$

$$\lg_3 n = k \lg_3 \frac{3}{2} = k(1 - \lg_3 2)$$

Puesto que $\lg_3 2 = 0{,}63092$, tenemos que

$$k = \frac{1}{1 - \lg_3 2} \lg_3 n = c \lg_3 n \quad con\ \ 2 < c < 3$$

Sustituyendo el valor de k en la ecuación de $t(n)$ obtenemos

$$t(n) = (a + \frac{b}{2})3^{c \lg_3 n} - \frac{b}{2} = (a + \frac{b}{2})n^c - \frac{b}{2}$$

siendo c una constante con valor $2 < c < 3$.

El algoritmo Orden_Por_Inserción es $O(n^2)$ y los algoritmos Orden_Por_Mezcla y Orden_Por_Montículo son $O(n \lg n)$, por lo tanto OrdenSolapa es asintóticamente peor.

El espacio extra empleado por OrdenSolapa corresponde a la pila de recursión. El tamaño máximo que alcanzará esa pila será el necesario para realizar

por completo una llamada recursiva (cuando se completa una llamada, la pila queda vacía).

Si denominamos $s(n)$ al espacio necesario para ORDENSOLAPA con una entrada de tamaño n, entonces $s(n)$ responde a la siguiente recurrencia:

$$s(n) = s(\frac{2n}{3}) + O(1)$$

que resolviendo de manera análoga a como hemos resuelto $t(n)$, obtenemos $s(n) = O(c \lg_3 n)$ con $2 < c < 3$. Por lo tanto, $s(n) = O(\lg n)$.

8. Un mínimo local de un array $A(a-1..b+1)$ es un elemento $A(k)$ que satisface $A(k-1) \geq A(k) \leq A(k+1)$. Suponemos que $a \leq b$ y $A(a-1) \geq A(a)$ y $A(b) \leq A(b+1)$; estas condiciones garantizan la existencia de algún mínimo local. Escriba un algoritmo que encuentre algún mínimo local de $A(a-1..b+1)$ y que sea sustancialmente más rápido que el evidente de O(n) en el caso peor. ¿De qué orden es su algoritmo?

   Solución:
   Podemos diseñar un algoritmo basado en la idea de la búsqueda dicotómica. Se comprueba si el punto central $\lfloor \frac{a+b}{2} \rfloor$ es un mínimo local; en caso afirmativo, ese es el resultado; si no lo es, debe ocurrir uno de estos dos casos:

   a) $A(\lfloor \frac{a+b}{2} \rfloor) \geq A(\lfloor \frac{a+b}{2} \rfloor + 1)$
   b) $A(\lfloor \frac{a+b}{2} \rfloor - 1) \leq A(\lfloor \frac{a+b}{2} \rfloor)$

   Si ocurre el caso (*a*) resolvemos el mismo problema original pero en el array $A(\lfloor \frac{a+b}{2} \rfloor..b+1)$, que es de tamaño mitad que el original.

   Si ocurre el caso (*b*) resolvemos análogamente, pero con el array $A(a-1..\lfloor \frac{a+b}{2} \rfloor)$.

   Lo importante es que basta con tomar una sola de las dos alternativas, ya que las condiciones garantizan la existencia de algún mínimo local en el subintervalo considerado.

   La función de coste temporal de este algoritmo será de la forma

$$f(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ f(\frac{n}{2}) + \Theta(1) & \text{si } n > 1 \end{cases}$$

   que resolviendo, obtenemos $f(n) = \Theta(\lg n)$.

9. Un vector T contiene n números naturales distintos. Queremos un algoritmo que *imprima* los m números menores de T. Sabemos que m es mucho menor que n. ¿Cómo lo haría para que resultara más eficiente?

   - Ordenar T usando ORDEN_POR_MEZCLA o bien ORDENACIÓN_RÁPIDA y luego imprimir los m primeros.
   - Usar algún otro método.

Estudie si cambiaría algo para los casos particulares: $m \in \Theta(\sqrt{n})$ y $m \in \Theta(\lg n)$.

Solución:
Ordenar con ORDEN_POR_MEZCLA es de $O(n \lg n)$, y hacerlo con ORDENACIÓN_RÁPIDA es de $O(n^2)$ en el caso peor y $O(n \lg n)$ en el caso medio.

Podemos considerar varios métodos alternativos. Por ejemplo:

a) Construir un montículo-min con los n valores de T; eso resulta de $\Theta(n)$. Después repetir m veces la retirada de la raíz del montículo; eso es de $O(m \lg n)$. Concluimos que este método sería $O(n + m \lg n)$.

b) Construir un montículo-max M con m elementos de T; eso resulta de $\Theta(m)$. Después realizar el procedimiento siguiente:

```
for i de los (n − m) restantes de T loop
   if i < RAÍZ(M) then
      M(1) ← i
      HUNDIR(M, m, 1)
```

Este procedimiento es de $O((n-m) \lg m)$, que en el caso de $m << n$ es igual a $O(n \lg m)$. Así pues, el método completo resulta de $O(m + n \lg m)$.

c) Invocar el procedimiento SELECCIONAR-K-ÉSIMO(T, m). Obsérvese que, como efecto secundario, los m menores elementos del vector T quedarán en las m primeras posiciones, aunque no estén ordenados de menor a mayor. En el caso peor, es de $O(mn)$; pero en el caso medio es $O(n)$.

Concluimos que (asumiendo $m << n$), en el caso peor, la alternativa (a) de $O(n + m \lg n)$ es la mejor. Pero en el caso medio, la alternativa (c) de $O(n)$ es la más eficiente.

En el caso particular de $m \in \Theta(\sqrt{n})$:

a) es $O(n + \sqrt{n} \lg n) = O(n)$.
b) es $O(\sqrt{n} + n \lg \sqrt{n}) = O(\sqrt{n} + n \lg n)$.
c) es $O(n\sqrt{n})$ en el caso peor y $O(n)$ en el caso medio.

Lo preferible en el caso peor es (a), y en el caso medio (a) o (c).

En el caso particular de $m \in \Theta(\lg n)$:

a) es $O(n + \lg n \lg n) = O(n)$.
b) es $O(\lg n + n \lg \lg n) = O(n \lg \lg n)$.
c) es $O(n \lg n)$ en el caso peor y $O(n)$ en el caso medio.

Lo preferible en el caso peor es (a), y en el caso medio (a) o (c).

10. Sea un vector $T(1 \ldots n)$ de números enteros, que representa n temperaturas tomadas en sucesivos intervalos de un minuto de tiempo. Diseñe un algoritmo que use la técnica *Divide y Vencerás* para encontrar el menor intervalo de

tiempo en el que todas las temperaturas registradas son de valor bajo cero. Es decir, si hay temperaturas bajo cero, queremos los índices a y b del intervalo $T(a \dots b)$ de menor número de elementos de $T(1 \dots n)$ tal que $(a \leq x \leq b \rightarrow T(x) < 0) \wedge (a > 1 \rightarrow T(a-1) \geq 0) \wedge (b < n \rightarrow T(b+1) \geq 0)$. Si no hay temperaturas bajo cero en $T(1 \dots n)$, basta una pareja de valores 0 para indicar esa circunstancia. ¿Cree que ese algoritmo es óptimo? Justifique su respuesta.

Solución:
Usando la técnica *Divide y Vencerás*, podemos diseñar el siguiente algoritmo. Dividimos el vector en dos mitades. El menor intervalo en el que se registraron temperaturas bajo cero estará incluido en la primera mitad o en la segunda mitad o tendrá, necesariamente, elementos del final de la primera mitad y del principio de la segunda mitad.

Hay que tener especial cuidado con los intervalos que son fronterizos con el centro del vector. Por ejemplo, en el vector $[1, 0, -1, -2, -1, 0, -3, -2, -2, -2, 1, 2, 3, 3]$, las dos mitades son $[1, 0, -1, -2, -1, 0, -3]$ y $[-2, -2, -2, 1, 2, 3, 3]$. El intervalo más pequeño de la primera mitad, que satisface las propiedades que buscamos, es $[-3]$ (formado por el último elemento de más a la derecha). Sin embargo, no podemos descartar el intervalo $[-1, -2, -1]$, aunque tenga tres elementos; porque al concatenar los intervalos fronterizos con el centro $[-3]$ y $[-2, -2, -2]$, para que se cumplan las condiciones que buscamos, resulta que el intervalo $[-1, -2, -1]$ es el más pequeño del vector dado inicialmente.

Lo que vamos a hacer es evitar, en las llamadas recursivas, los dos elementos que hacen frontera con el centro del vector. Diseñaremos un procedimiento particular para el tratamiento de esos valores fronterizos y sus adyacentes negativos.

En un intervalo determinado por los índices i y j, asumiendo que hay al menos dos elementos, los elementos fronterizos con el centro de ese intervalo son los que ocupan los índices $\lfloor \frac{i+j}{2} \rfloor$ y $\lfloor \frac{i+j}{2} \rfloor + 1$. El algoritmo INTERVALO_CENTRAL$(T,\ i,\ j)$ calcula el intervalo de negativos, que satisface las condiciones que buscamos, del vector $T(i..j)$ que incluye a alguno de esos elementos fronterizos. Si no existe tal intervalo, devuelve el valor estructurado especial $(0, 0, \infty)$, que representa dos índices artificiales 0 y una longitud de intervalo imposible ∞.

pre: En $T(i..j)$ hay al menos dos elementos $(i + 1 \leq j)$.
func INTERVALO_CENTRAL$(T,\ i,\ j)$ **return** $(Int \times Int \times Int)$
  $centro \leftarrow \lfloor \frac{i+j}{2} \rfloor$
  $izq \leftarrow centro$
  while $(i \leq izq) \wedge (T(izq) < 0)$ **loop**
    $izq \leftarrow izq - 1$
  end loop
  // $T(izq) \geq 0$. El intervalo empezaría en $izq + 1$, si existe.
  $der \leftarrow centro + 1$
  while $(der \leq j) \wedge (T(der) < 0)$ **loop**
    $der \leftarrow der + 1$

end loop
// $T(der) \geq 0$. El intervalo terminaría en $der - 1$, si existe.
if $izq + 1 > der - 1$ **then** //no hay intervalo que satisfaga las condiciones
  return $(0, 0, \infty)$
else
  return $(izq + 1,\ der - 1,\ der - izq - 1)$

Para resolver el problema que nos piden, diseñaremos un algoritmo recursivo que resuelve directamente los casos de vectores de longitud menor o igual que 1; y para los vectores de longitud mayor o igual que 2:

- resuelve recursivamente el problema para la primera mitad —excluyendo el elemento fronterizo de la derecha—,
- resuelve recursivamente el problema para la segunda mitad —excluyendo el elemento fronterizo de la izquierda—,
- calcula el intervalo de temperaturas bajo cero que satisface las condiciones e incluye alguno de los elementos fronterizos y
- finalmente se queda con el más pequeño de esos intervalos, si existe alguno.

Si hay temperaturas bajo cero, MIN_INTERVALO_HEL(T, i, j) devuelve los índices del intervalo buscado, incluido en el vector $T(i \ldots j)$, y además el número de elementos de ese intervalo. En el caso de que no haya temperaturas bajo cero en $T(i \ldots j)$, devuelve dos índices artificiales de valor 0, indicando que no hay intervalo que satisfaga las condiciones, y además un valor artificial ∞, suficientemente grande, que se elige así para facilitar la codificación del algoritmo. Obsérvese que si hay temperaturas bajo cero, el valor ∞ siempre será mayor que cualquier número de elementos del intervalo.

func MIN_INTERVALO_HEL$(T,\ i,\ j)$ **return** $(Int \times Int \times Int)$
  if $i > j$ **then return** $(0, 0, \infty)$
  elsif $i = j$ **then**
    if $T(i) < 0$ **then return** $(i,\ j,\ 1)$
    else return $(0,\ 0,\ \infty)$
  else
    $(a1,\ b1,\ cont1) \leftarrow$ MIN_INTERVALO_HEL$(T,\ i,\ \lfloor \frac{i+j}{2} \rfloor - 1)$
    $(a2,\ b2,\ cont2) \leftarrow$ MIN_INTERVALO_HEL$(T,\ \lfloor \frac{i+j}{2} \rfloor + 2,\ j)$
    $(a3,\ b3,\ cont3) \leftarrow$ INTERVALO_CENTRAL$(T,\ i,\ j)$
    if $cont1 \leq cont2 \wedge cont1 \leq cont3$ **then**
      return $(a1,\ b1,\ cont1)$
    elsif $cont2 \leq cont1 \wedge cont2 \leq cont3$ **then**
      return $(a2,\ b2,\ cont2)$
    else
      return $(a3,\ b3,\ cont3)$

La función de coste temporal de MIN_INTERVALO_HEL($T,\ 1,\ n$) responde a la recurrencia siguiente:

$$f(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ 2f(\frac{n}{2}) + \Theta(n) & \text{si } n > 1 \end{cases}$$

y resolviendo obtenemos $f(n) = \Theta(n \lg n)$.

Evidentemente, este algoritmo no es óptimo. Obsérvese que basta un recorrido secuencial del vector $T(1 \ldots n)$, en el que vayamos guardando los índices del menor intervalo de valores negativos, para resolver el problema. Obviamente eso puede hacerse en $\Theta(n)$.

11. El *diámetro* de un árbol es la longitud del camino más largo entre cualquier par de nodos. Diseñe un algoritmo, lineal en el número de nodos, que calcule el *diámetro* de un árbol.

Solución:
Presentamos la guía de solución sólo para el caso de un árbol binario. Habría que generalizarla para el caso de un árbol general.

Si a es un árbol binario, denotamos $a.izq$ y $a.der$ a sus subárboles izquierdo y derecho, respectivamente, si es que existen.

Un algoritmo lineal para resolver este problema puede basarse en la siguiente especificación recursiva de la función *diámetro*, que utiliza la función *alt* que calcula la *altura* de un árbol:

$$diám(a) = \begin{cases} 0 & \text{si } a.der = nil \wedge a.izq = nil \\ \mathbf{max}\{diám(a.izq), & \\ alt(a.izq) + 1\} & \text{si } a.der = nil \wedge a.izq \neq nil \\ \mathbf{max}\{diám(a.der), & \\ alt(a.der) + 1\} & \text{si } a.der \neq nil \wedge a.izq = nil \\ \mathbf{max}\{diám(a.izq), & \\ alt(a.izq) + alt(a.der) + 2, & \\ diám(a.der)\} & \text{si } a.izq \neq nil \ \wedge \ a.der \neq nil \end{cases}$$

La implementación puede hacerse lineal en el número de nodos del árbol ya que hay exactamente una llamada recursiva por cada nodo del árbol; y el número de operaciones que hay que realizar, una vez resueltas las llamadas recursivas, es constante.

La función *alt(a)* queda especificada por la ecuación siguiente:

$$alt(a) = \begin{cases} 0 & \text{si } a.der = nil \wedge a.izq = nil \\ alt(a.izq) + 1 & \text{si } a.der = nil \wedge a.izq \neq nil \\ alt(a.der) + 1 & \text{si } a.der \neq nil \wedge a.izq = nil \\ \mathbf{max}\{alt(a.izq) + 1 & \\ alt(a.der) + 1\} & \text{si } a.izq \neq nil \ \wedge \ a.der \neq nil \end{cases}$$

y su cálculo puede integrarse dentro del mismo algoritmo que calcule *diámetro*, o bien calcularse aparte, sin que se produzca incremento del coste asintótico.