Fuzzy

Algorítmica

Capítulo 2: Algoritmos Divide y Vencerás

Solución a los ejercicios de la relación de prácticas

Objetivos de las prácticas de la relación Fuzzy

1. Apreciar la utilidad de la técnica Divide y Vencerás para resolver problemas de forma más eficiente que otras alternativas más sencillas o directas.
2. Comprobar la utilidad de las Ecuaciones Recurrentes en Algorítmica.
3. Constatar el buen funcionamiento de la técnica en distintos contextos.
4. Trabajar de forma autónoma o en equipo.
5. Se sugieren las vías de solución de los 5 problemas propuestos, pero el trabajo final tiene que hacerlo cada cual (solo o en compañía), y
6. Demostrarnos a nosotros mismos que sabemos expresar en público las ventajas, inconvenientes y alternativas empleadas, para lograr la solución alcanzada

Cada elemento en su posición

Fuzzy

- Dado un vector V de números enteros, todos distintos, ordenado de forma no decreciente, se quiere determinar si existe un índice i tal que $V[i] = i$ y encontrarlo en ese caso.
- Diseñar e implementar un algoritmo Divide y Vencerás que permita resolver el problema. ¿Que complejidad tiene ese algoritmo? ¿Y el algoritmo “obvio” para realizar esa tarea?
- Supóngase ahora que los enteros no tienen por que ser todos distintos (pueden repetirse). Determinar si el algoritmo anterior sigue siendo válido, y en caso negativo proponer uno que si lo sea. ¿Sigue siendo preferible al algoritmo obvio?

Cada elemento en su posición

- Como el vector está ordenado de menor a mayor, podemos actuar como con la búsqueda binaria, examinando el elemento que se encuentra en la posición de la mitad, m = (n + 1)/2 (la mediana en este caso).
- Si coincide para ese elemento su valor con el índice, es decir si v[m] = m, ya hemos encontrado el índice buscado.
- En caso contrario, si el valor de ese entero es mayor que el índice (v[m] > m), sabemos que para todos los índices mayores que m, los valores en esas posiciones serán siempre mayores que los propios índices, es decir v[j] > j, ∀j > m
- Por tanto sabemos que en el lado derecho del vector, a partir de la posición m, no se puede producir la situación buscada.

Cada elemento en su posición

- Basta entonces comprobar si en la parte izquierda del vector se puede producir tal situación.
- Reducimos la búsqueda entonces al subvector desde la posición inicial a la posición m − 1.
- Si lo que ocurre es que v[m] < m, entonces el razonamiento es el mismo pero al revés
- Esto da lugar al siguiente algoritmo DV:

```
localiza(v,primero,ultimo) {
  if (primero==ultimo)
  then if v[primero]==primero) then return primero;
       else return 0 //no existe el indice buscado
  else {
        i=(primero+ultimo)/2; //division entera
        if (v[i]=i) then return i;
        else if (v[i]>i) then return localiza(v,primero,i-1);
             else return localiza(v,i+1,ultimo);
}
```

Cada elemento en su posición

- No nos entretendremos en demostrar la propiedad clave para el diseño del algoritmo: PRADO
- Resumidamente:

  Que si $v[m] > m$ entonces $v[j] > j, \forall j > m$, se puede demostrar por inducción.

  Para el caso base, al ser $v[m + 1] > v[m]$, entonces

  $$v[m + 1] \geq v[m] + 1 > m + 1$$

  Para el paso de inducción, si $v[j] > j$, entonces (por la misma razón de antes) $v[j + 1] \geq v[j] + 1 > j + 1$.

Cada elemento en su posición

- Cuando los enteros se pueden repetir el algoritmo anterior puede no funcionar correctamente (en la demostración de la propiedad clave es preciso suponer que v[j+1] > v[j], no bastaba con v[j+1] ≥ v[j]).
- Por ejemplo para el vector

i	1	2	3	4	5	6	7
$v[i]$	2	3	4	6	6	6	9

- Resulta evidente que v[6] = 6, pero el algoritmo anterior no detectaría este hecho y devolvería 0.
- El algoritmo anterior puede fallar porque no podemos asegurar, cuando por ejemplo v[m] > m, que podamos descartar totalmente la parte derecha del vector a partir de m. Esto da lugar a un algoritmo DyV que puede hacer 2 llamadas recursivas en cada caso en lugar de solo una como ocurría antes

Mezcla de k vectores ordenados

- Se tienen *k* vectores ordenados (de menor a mayor), cada uno con *n* elementos, y queremos combinarlos en un único vector ordenado (con $k \times n$ elementos).
- Una alternativa directa, utilizando un algoritmo clásico, es mezclar los dos primeros vectores, posteriormente mezclar el resultado con el tercero, y así sucesivamente.
  - ¿Cuál sería la eficiencia de este algoritmo?
  - Diseñar, analizar la eficiencia e implementar un algoritmo de mezcla mas eficiente, basado en Divide y Vencerás.

- El algoritmo obvio haría uso de un método para mezclar dos vectores ordenados (con un tiempo de ejecución proporcional a la suma de los tamaños de los vectores que se mezclan)
- Por tanto, en mezclar los dos primeros vectores tardaría un tiempo n + n. Para mezclar ese vector con el 3º tardaría 2n+n, y así sucesivamente, de modo que para mezclar el vector resultante con el k-ésimo (el último) tardaría (k−1)n+n.
- Por tanto el tiempo total de ejecución es

$$\sum_{i=1}^{k-1}(in+n) = n\sum_{i=1}^{k-1} i + n\sum_{i=1}^{k-1} 1 = n\frac{k(k-1)}{2} + (k-1)n = n\frac{(k-1)(k+2)}{2}$$

- Es decir, cuadrático en el número de vectores a mezclar: $O(nk^2)$

Mezcla de k vectores ordenados

- Descompongamos el problema
- Por ejemplo, si $k = 2^m$, entonces podríamos mezclar las $k/2 = 2^{m-1}$ parejas de vectores (de longitud n) (el vector 1 con el 2, el vector 3 con el 4, hasta el vector $k - 1$ con el k), luego mezclar también las $k/4 = 2^{m-2}$ parejas de vectores (de longitud 2n) (el vector 1-2 con el 3-4, el 5-6 con el 7-8,...), y así sucesivamente hasta mezclar la última pareja resultante de vectores (de longitud $2^{m-1}n = nk/2$).
- Este proceso, en cada iteración mezcla $k/2^i = 2^{m-i}$ parejas de vectores de tamaño $2^{i-1}n$, tardando pues un tiempo proporcional a $2^{m-i}\,2\,2^{i-1}n = 2^m n = kn$.
- Como se realizan $m = \log k$ iteraciones, el tiempo total es proporcional a $kn \log k$.

Mezcla de k vectores ordenados

- El proceso consistiría en dividir el problema de mezclar k vectores en dos subproblemas de mezclar k/2 vectores y luego mezclar los dos vectores resultantes (cada uno de tamaño nk/2)
- Como esta mezcla puede hacerse en tiempo

$$nk/2 + nk/2 = nk$$

entonces el tiempo de ejecución de este algoritmo sería

$$T(k) = 2T(k/2) + nk.$$

- De esta recurrencia se deduce un tiempo de O(nk log k).
- El caso base del algoritmo sería cuando k = 1, en cuyo caso el procedimiento devolvería el mismo vector.