Algorítmica

Capítulo 2: Algoritmos Divide y Vencerás

Ejercicios prácticos

Objetivos de las prácticas

- El objetivo de esta práctica es múltiple
  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. Constatar el mejor funcionamiento de la técnica conforme menos sub-problemas hay y mas parecidos en tamaño son estos.
  3. Comprobar la utilidad de las Ecuaciones Recurrentes en Algorítmica.
  4. Trabajar comprometidamente en equipo
  5. Aprender a expresar en público las ventajas, inconvenientes y alternativas empleadas, para lograr la solución alcanzada

Comparación de preferencias: Planteamiento

- Muchos sitios web intentan comparar las preferencias de dos usuarios para realizar sugerencias a partir de las preferencias de usuarios con gustos similares a los nuestros.
- Dado un ranking de *n* productos (p.ej. películas) mediante el cual los usuarios indicamos nuestras preferencias, un algoritmo puede medir la similitud de nuestras preferencias contando el número de inversiones: dos productos *i* y *j* están "invertidos" en las preferencias de A y B si el usuario A prefiere el producto *i* antes que el *j*, mientras que el usuario B prefiere el producto *j* antes que el *i*. Esto es, cuantas menos inversiones existan entre dos rankings, más similares serán las preferencias de los usuarios representados por esos rankings.

Comparación de preferencias: Planteamiento

- Por simplicidad podemos suponer que los productos se pueden identificar mediante enteros 1, . . . , n, y que uno de los rankings siempre es 1, . . . , n (si no fuese así bastaría renumerarlos) y el otro es $a_1, a_2, \ldots, a_n$, de forma que dos productos i y j están invertidos si $i < j$ pero $a_i > a_j$.
- De esta forma nuestra representación del problema será un vector de enteros V de tamaño n, tal que $V[i] = a_i$, $i = 1,\ldots,n$
- Se pide diseñar, analizar la eficiencia e implementar un algoritmo Divide y Vencerás para medir la similitud entre dos rankings. Compararlo con el algoritmo de “fuerza bruta” obvio y realizar un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Comparación de preferencias: Solución

- Un algoritmo simple (Fuerza Bruta)
- Un algoritmo de fuerza bruta simplemente comparara las preferencias de los usuarios probando todos los pares posibles de productos, con un coste $O(n^2)$.
- Podría ser el siguiente:

```
inv=0;
for i=1 to n-1
        for j=i+1 to n
                if (v[i]> v[j]) inv=inv+1;
return (inv);
```

Comparación de preferencias: Solución Divide y Vencerás

- Otra forma de enfocar el problema es darse cuenta de que el número de inversiones es igual al número de trasposiciones necesarias para ordenar el vector, es decir el número mínimo de cambios entre dos posiciones (intercambiar los valores de v[i] y v[j]) que hay que hacer para que se verifique que v[i] = i.
- Por ejemplo, para el vector

i	1	2	3	4	5
v[i]	3	1	4	2	5

- el número de inversiones es 3 (pares 3-1, 3-2 y 4-2), que coincide con el número de trasposiciones para ordenar el vector, como se ve en la siguiente tabla

i	1	2	3	4	5
v[i]	3	1	4	2	5
1	1	3	4	2	5
2	1	3	2	4	5
3	1	2	3	4	5

Comparación de preferencias: Solución Divide y Vencerás

- Podemos pensar entonces en emplear algún método similar al algoritmo de ordenación por fusión (mergesort): (1) dividir la lista de productos (el vector v) en dos mitades y contar (recursivamente) el número de inversiones en cada mitad; (2) para combinar esos resultados y obtener el resultado final será necesario sumar esas dos cantidades, mas una tercera que será el recuento de inversiones en las que a_i y a_j están en mitades diferentes.
- Por ejemplo, para el vector

i	1	2	3	4	5	6	7	8
v[i]	6	1	4	2	3	8	7	5

- el número de inversiones es 10 (pares 6-1, 6-4, 6-2, 6-3, 6-5, 4-2, 4-3, 8-7, 8-5, 7-5). Las inversiones de la mitad izquierda son 4 (6-1, 6-4, 6-2, 4-2), las de la mitad derecha son 3 (8-7, 8-5, 7-5), y las inversiones que implican elementos de las dos partes son 3 (6-3, 6-5, 4-3).
- Por tanto podemos construir un algoritmo DyV mejor que el método de fuerza bruta si conseguimos realizar eficientemente la tarea de contar el número de inversiones que implican elementos de mitades diferentes.

Comparación de preferencias: Solución Divide y Vencerás

- Si supusiéramos que ambas mitades están ordenadas, entonces podríamos utilizar un procedimiento similar al de mezcla del algortimo MergeSort para contar eficientemente el número de inversiones, y de paso ordenar el vector resultante de unir las dos mitades.
- El procedimiento podría ser:

```
cuenta_inversiones(v)  {
  n=length.v;
  if (n==1)
        return (0,v);
  dividir el vector v en dos mitades, a y b;
  (ri,a)=cuenta_inversiones(a);
  (rd,b)=cuenta_inversiones(b);
  (r,v)=mezcla_y_cuenta(a,b);
  return (ri+rd+r,v);
}
```

Comparación de preferencias: Solución Divide y Vencerás

El procedimiento mezcla y cuenta sería el siguiente:

```
mezcla_y_cuenta(a,b) {
  i=1; j=1; k=1;
  inv=0;
  fin=false;
  while (not fin) {
    if (a[i]<b[j]) {
      v[k]=a[i];
      k=k+1;
      if (i==length.a) {
        fin=true;
        for h=k to length.a+length.b {
            v[h]=b[j];
            j=j+1;
        }
      }
      else i=i+1;
    }
    else {
      v[k]=b[j];
      k=k+1;
      inv=inv+length.a-i+1;
      if (j==length.b) {
        fin=true;
        for h=k to length.a+length.b {
            v[h]=a[i];
            i=i+1;
        }
      }
      else j=j+1;
    }
  }
  return (inv,v);
}
```

Comparación de preferencias: Solución Divide y Vencerás

- El procedimiento mezcla y cuenta recorre una sola vez cada una de las dos mitades, por lo que su complejidad es O(n).
- Por tanto el procedimiento cuenta inversiones tiene una ecuación de recurrencia del tipo

$$T(n) = 2T(n/2) + O(n)$$

- que ya sabemos que corresponde a una complejidad de O(n log n).

Eliminar elementos repetidos

- Dado un vector de n elementos, de los cuales algunos pueden estar duplicados, el problema es obtener otro vector donde todos los elementos duplicados hayan sido eliminados.
- Diseñar, analizar la eficiencia e implementar un algoritmo sencillo para esta tarea, y luego hacer lo mismo con un algoritmo más eficiente, basado en Divide y Vencerás, de con eficiencia $O(n \log n)$.
- Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Eliminar elementos repetidos

- Un posible algoritmo consistiría en ir comparando cada elemento del vector v con los demás, y si son iguales eliminar uno de ellos. Los elementos no duplicados se van guardando en otro vector x. El algoritmo sería:

```
x[1]=v[1];
k=1;
for i=2 to n {
    j=1;
    found = false;
    while (j<=k && !found) {
        if (v[i]==x[j]) then found=true;
        else j=j+1;
    }
    if (!found) {
        k=k+1;
        x[k]=v[i];
    }
}
```

- Pero este proceso tiene un tiempo de ejecución $O(n^2)$ que es mayor que el exigido.

Eliminar elementos repetidos

- Otra alternativa, si no es importante que los elementos no duplicados permanezcan en el mismo orden original, sería primero ordenar el vector v de menor a mayor (lo que se puede hacer en un tiempo O(n log n)). Con ello nos aseguramos que todos los elementos iguales entre sí están contiguos, lo que nos permite que en una sola pasada sobre el vector ordenado se puedan detectar y eliminar los duplicados.
- El algoritmo es:

```
Ordenar el vector v en orden no decreciente;
x[1]=v[1];
k=1;
for i=2 to n {
    if (v[i] != x[k]) then {
        k=k+1;
        x[k]=v[i];
    }
}
```

- Y como el recorrido de v se hace en tiempo O(n), el tiempo total del algoritmo sigue siendo O(nlog n)

Eliminar elementos repetidos

- La mejor alternativa es emplear un enfoque DV directamente (no a través de un algoritmo de ordenación basado en DV):
- Dividimos el problema en 2 subproblemas de tamaño mitad app, eliminamos los duplicados en cada una de las mitades, y luego eliminamos los elementos duplicados en las dos mitades.
- Si suponemos que los subproblemas se resuelven eliminado duplicados y ordenándolos de menor a mayor, entonces es posible diseñar un método eficiente para componer las soluciones de los dos subproblemas (que también ordena el resultado de menor a mayor):

```
elimina_duplicados(v) {

n=length.v;
 if (n==1)
    return (v);
 dividir el vector v en dos mitades, a y b;
 a=elimina_duplicados(a);
 b=elimina_duplicados(b);
 v=mezcla_y_elimina(a,b);
 return (v);
}
```

Eliminar elementos repetidos

- El procedimiento mezcla_y_elimina sería así:

```
mezcla_y_elimina(a,b) {
 i=1; j=1; k=1;
 fin=false;
 while (not fin) {
    if (a[i]<b[j]) {
      v[k]=a[i];
      k=k+1;
      if (i==length.a) {
        fin=true;
        for h=j to length.b {
           v[k]=b[h];
           k=k+1;
        }
      }
      else i=i+1;
    }
    else if (a[i]>b[j]) {
          v[k]=b[j];
          k=k+1;
          if (j==length.b) {
            fin=true;
            for h=i to length.a {
               v[k]=a[h];
               k=k+1;
            }
          }
          else j=j+1;
    }
    else { //coinciden a[i] y b[j]
          v[k]=a[i];
          k=k+1;
          if (i==length.a) {
            fin=true;
            for h=j+1 to length.b {
              v[k]=b[h];
               k=k+1;
            }
          }
          else { i=i+1; j=j+1;}
          if (j==length.b) {
            fin=true;
            for h=i+1 to length.a {
               v[k]=a[h];
               k=k+1;
            }
          }
          else{ i=i+1; j=j+1;}
    }
 }
 return (v);
}
```

Eliminar elementos repetidos

- El procedimiento mezcla y elimina recorre una sola vez cada una de las dos mitades, por lo que su complejidad es O(n).
- Por tanto el tiempo de ejecución del procedimiento elimina duplicados tiene una ecuación de recurrencia del tipo

$$T(n) = 2T(n/2) + O(n)$$

que ya sabemos le corresponde una complejidad de O(n log n).

Cada elemento en su posición

- 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 Vencerá que permita resolver el problema. ¿Que complejidad tiene ese algoritmo? ¿Y el algoritmo "obvio" para realizar esa tarea? Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.
- 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 (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), al estar los elementos ordenados y no repetirse, 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 (esto se demuestra más adelante).
- 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: no se puede producir la situación buscada en la parte izquierda del vector (desde el inicio hasta m), y solo tenemos que comprobar si ocurre en la parte derecha, desde la posición m + 1 hasta la final.
- 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

- Demostración de la propiedad clave para el diseño del algoritmo

  El hecho de que si $v[m] > m$ entonces $v[j] > j \ \forall j > m$, se puede demostrar fácilmente por inducción. Para el caso base, al ser $v[m + 1] > v[m]$ (por estar ordenado el vector y no repetirse los enteros), 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) $v[j + 1] \geq v[j] + 1 > j + 1$.

- Complejidad

  El tiempo de ejecución del algoritmo viene definido por la expresión $T(n) = T(n/2) + a$.

  Por tanto el algoritmo tiene una complejidad de $O(\log n)$.

Cada elemento en su posición

- Cuando los enteros se pueden repetir el algoritmo anterior puede no funcionar correctamente (nótese que en la demostración de la propiedad clave era 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 (y por tanto un algoritmo correcto debería devolver 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.

Cada elemento en su posición

- Pero lo que si se puede asegurar en ese caso es que para ningún índice j situado entre los índices m y v[m] − 1 puede ser que v[j] = j.
- Esto es así porque para cualquier j en ese rango v[j] ≥ v[m] > j.
- Por tanto podemos centrarnos (además de en el rango de 1 a m − 1) en el rango de v[m] a n.
- En caso de que sea v[m] < m el razonamiento es similar:
  - Para los índices j en el rango entre v[m] + 1 y m tenemos v[j] ≤ v[m] < j, y por tanto podemos centrarnos en los rangos entre m + 1 y n y entre 1 y v[m].
- Esto da lugar a un algoritmo DV que puede hacer dos llamadas recursivas en cada caso, en lugar de una sola como ocurría antes.

Cada elemento en su posición

```
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 {
                   izq=localiza(v,primero,i-1);
                   if (v[i]<=ultimo) then der=localiza(v,v[i],ultimo);
                                     else der=0;
                   if (izq != 0) then return izq;
                                 else if (der != 0) then return der;
                                      else return 0;
             }
             else {
                  der=localiza(v,i+1,ultimo);
                  if (primero<=v[i]) then izq=localiza(v,primero,v[i]);
                                     else izq=0;
                  if (der != 0) then return der;
                                else if (izq != 0) then return izq,
                                     else return 0;
             }
}
```

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×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.
  - Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Mezcla de k vectores ordenados

- El algoritmo inicialmente propuesto haría uso de un método para mezclar dos vectores ordenados cuyo tiempo de ejecución es 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, para mezclar el resultado con el 4º tardaría 3n + 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

- Otra forma de proceder para resolver este problema sería pensando en descomponer el problema. Por ejemplo, si suponemos por un momento que k es una potencia de 2, $k = 2^m$, 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

- Más formalmente, 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), empleando el algoritmo clásico de mezcla.
- 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 (un solo vector), en cuyo caso el procedimiento devolvería el mismo vector.

Serie unimodal de números

- Sea un vector V de números de tamaño *n*, todos distintos, de forma que existe un índice *p* (que no es ni el primero ni el último) tal que a la izquierda de *p* los números están ordenados de forma creciente y a la derecha de *p* están ordenados de forma decreciente:

$$\forall i, j \leq p,\ i < j \Rightarrow V[i] < V[j]$$

$$\forall i, j \geq p,\ i < j \Rightarrow V[i] > V[j]$$

- (de forma que el máximo se encuentra en la posición *p*).
- Diseñar un algoritmo Divide y Vencerás que permita determinar *p*. ¿Cuál es la eficiencia del algoritmo? Compararlo con el algoritmo "obvio" para realizar esta tarea.
- Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Serie unimodal de números

- Lo que haremos será dividir el problema en dos: accedemos al elemento que se encuentra en la posición mitad del vector para ver si el pico se encuentra antes o después de esa posición. Para ello miramos también los valores en las posiciones inmediatamente anterior e inmediatamente posterior y comparamos esos valores.
- Si el mayor de ellos es el que está en medio, sabemos que ese es el elemento buscado (es el único que cumple esa condición).
- En caso contrario, si el mayor es el posterior, sabemos que nos encontramos a la izquierda del "pico", por lo que buscaremos en la mitad derecha.
- Si el mayor es el anterior, estamos a la derecha del "pico" y por tanto buscamos en la mitad izquierda.

Serie unimodal de números

- El anterior razonamiento da lugar al siguiente algoritmo DV (en el que suponemos que n vale al menos 3)

```
unimodal(v,primero,ultimo) { //la llamada inicial sera unimodal(v,0,n-1)

 i=(primero+ultimo)/2; //division entera
 if ((v[i]>v[i-1]) && (v[i]>v[i+1])) then return i;
 else if (v[i]<v[i+1]) then return unimodal(v,i+1,ultimo);
      else return unimodal(v,primero,i-1); //en este caso ya sabemos que tiene que
                                           //ser v[i]<v[i-1]
}
```

- El tiempo de ejecución del algoritmo viene definido por la recurrencia

$$T(n) = T(n/2) + a$$

- Y por tanto T (n) es O(log n).