Algoritmia/Divide y vencerás

El primer método algorítmico presentado en este libro es la técnica de **divide y vencerás**, también conocida como divide y conquista (del inglés *divide and conquer*). El método se basa en reducir un problema dado en dos o más subproblemas más pequeños y, sucesivamente, volver a aplicar el mismo método sobre los resultados obtenidos hasta alcanzar subproblemas de resolución trivial o casos base. De manera general, estos son los pasos a seguir a la hora de crear un algoritmo basado en el método divide y vencerás:

1. Dado un problema, identificar los subproblemas del mismo tipo
2. Resolver recursivamente cada subproblema
3. Combinar las soluciones obtenidas en una única solución al problema inicial

Ejemplo: Mergesort

El pseudocódigo de la ordenación por fusión (en C Sharp)

```
int[] OrdenacionMerge(int []vector) {
    if (vector.Length==1) //Condición de fin de la recursividad (vector con un solo elemento)
        return vector;
    int medio=vector.Length/2; //Posición del centro de la lista
    //1. Creamos dos listas
    int[] lista1=new int[medio];
    int[] lista2=new int[vector.Length-medio];
    //2. Copiamos los elementos del vector, la mitad en lista1, y la otra mitad en lista2
    for (int i=0;i<medio;i++) lista1[i]=vector[i];
    for (int i=medio;i<vector.Length;+++) lista2[i-medio]=vector[i];
    //3. Invocamos recursivamente a la función
    int [] res1=OrdenacionMerge(lista1);
    int [] res2=OrdenacionMerge(lista2);
    int [] resultadofinal=vector; //Reutilización de vector de entrada
    //5.Fusionamos la lista
    int n1=0,n2=0;//Usamos dos contadores, n1 y n2 que avanzan en lista1 y lista2.
    for (int i=0;i<vector.Length;i++) {
        if (n1==res1.Length) { //Ya no hay más elementos en res1
            resultadofinal[i]=res2[n2];
            ++n2;
        } else if (n2==res2.Length) { //Ya no hay más elementos en res2
            resultadofinal[i]=res1[n1];
            ++n1;
        } else { //Hay elementos en ambas listas
            //Comparamos los elementos en las posiciones a analizar.
            if (res1[n1]<res2[n2]) {
                //Si el menor esta en res1 utilizar el mismo
                resultadofinal[i]=res1[n1];
                ++n1;
            } else {
                //Si el menor (o igual) está en res2 utilizar el mismo
                resultadofinal[i]=res2[n2];
                ++n2;
```

```
            }
        }
    }
    //6.Devolvemos el resultado final
    return resultadofinal;
}
```

Problemas resueltos

Subvector de suma máxima

- **Enunciado del problema**: "Dado un vector de longitud n, se desea encontrar el subvector de longitud m y suma máxima, con m ≤ n".
- **Solución**: Un primer acercamiento podría consistir en dividir el vector por la mitad y considerar los subvectores izquierdo y derecho. Sin embargo, esta solución no sería correcta al no considerarse los subvectores que atraviesan la mitad del vector. Por tanto, habrá que considerar los tres casos: subvector izquierdo, subvector derecho y subvector central.
  - **Subvector izquierdo**: Lo primero que se ha de hacer es comparar la longitud del vector con la del subvector; si la longitud del primero es mayor, se disminuye su tamaño a la mitad y se vuelve a aplicar el método, esta vez para calcular el subvector izquierdo. Por tanto, el coste de la llamada recursiva es $T(\frac{n}{2})$, siendo n la longitud del vector. En el momento en que la longitud del vector sea menor que la del subvector, se suman todas las posiciones consecutivas hasta llegar al final y se asignan los límites del subvector (inicio y fin). Es posible que dicho subvector tenga menos longitud que la deseada; por ello, cuando acaba la llamada para el subvector izquierdo, es necesario aumentar su longitud por la derecha hasta llegar a la longitud pedida. Según se aumenta la longitud, también habrá que ir actualizando la suma total del subvector.
  - **Subvector derecho**: Inmediatamente después de haber calculado el subvector izquierdo, se hará exactamente lo mismo con el subvector derecho, ya que las anteriores llamadas han ido dividendo el vector en dos. El coste para la llamada del subvector derecho también es $T(\frac{n}{2})$. Igual que antes, cuando se haya terminado de calcular el subvector derecho, es posible que este no tenga la longitud pedida, por lo cual se aumenta su longitud (esta vez por la izquierda) hasta llegar a la longitud pedida. Al finalizar el proceso, se actualiza su suma.
  - **Subvector central**: Lo primero que se hará será dividir el vector por la mitad. Acto seguido, se comprueba hasta dónde se puede calcular con la longitud que se está pidiendo, teniendo en cuenta que el subvector tiene que pasar por el punto medio del vector. Para ello se tienen dos marcadores, inicio y final, que van a indicar hasta que posiciones del vector hay que moverse para calcular un subvector con la longitud que se pide y que pase por el centro del vector. Puede darse el caso de que tanto inicio como final indiquen posiciones inexistentes en el vector si la longitud pedida supera la longitud de la mitad del vector. Por esta razón, si cualquiera de los dos se saliera del rango, ya sea inicio por abajo o final por arriba, se inicializarán como los extremos que les correspondan del vector: inicio = 0, final = n-1 (esta última no será necesaria).

    Si final se sale del rango, solo se podrán calcular n-l+1 (siendo l la longitud del subvector pedido) subvectores de longitud l, o lo que es lo mismo, solo se podrán calcular subvectores con inicio entre 0 y n-l. La forma de calcular los vectores máximos es mediante iteración. Cuando se acaba con un subvector, se compara su suma con la suma máxima para ver si hay que actualizar el vector óptimo.

    En caso de que final no se salga del rango se hace lo mismo que en el caso anterior, pero sin riesgo de evaluar un subvector inexistente.

- **Algoritmo**:

```
tipos
   vector : reg
               ini : ent
               fin : ent
               suma : ent
            freg
ftipos
```

```
fun subvector_optimo(v:vector, l:ent, c:ent, f:ent) dev vector
   // Variables locales
   izq, der, cent : vector  //Se divide el vector en tres subvectores
   optimo : vector          //valor devuelto por el metodo
   m, i : ent
   suma : ent
   tamaño_izq, tamaño_der : ent // tamaños del subvector izquierdo y derecho respectivamente

   suma := 0

   /* Se comprueba si el vector tiene mayor longitud que el subvector que se está buscando;
      si la longitud del vector es menor que la del subvector entonces se suman todas las
      posiciones hasta el final del vector */

   si (f-c+1 <= l) entonces
      para i:= c hasta f hacer              // coste lineal
         suma := suma + v[i]
      fpara
      optimo.ini := c                       // operaciones con coste cte
      optimo.fin := f
      optimo.suma := suma
   si_no                                    // el vector tiene mayor longitud que la solicitada
      m := (c+f) / 2
      izq := subvector_optimo (v, l, c, m)  // coste T(n/2)
      /* Si el vector izquierdo es de menor longitud que l entonces se amplía por la derecha
         hasta llegar a la longitud l pasada por parámetro*/
      tamaño_izq := izq.fin - izq.ini + 1
      si (tamaño_izq < l) entonces
         mientras (tamaño_izq < l) hacer
            izq.fin := izq.fin + 1;
            izq.suma := izq.suma + v[izq.fin]
            tamaño_izq := tamaño_izq + 1
         fmientras
      fsi

      der := subvector_optimo (v, l, m+1, f);    //coste T(n/2)
      /* Si el vector derecho es de menor longitud que l entonces se amplía por la izquierda
         hasta llegar a la longitud l pasada por parámetro*/
```

```
      tamaño_der := der.fin - der.ini + 1
      si (tamaño_der < l) entonces
         mientras (tamaño_der < l) hacer
            der.ini := der.ini - 1
            der.suma := v[der.ini]
            tamaño_der := tamaño_der + 1
         fmientras
      fsi
      cent := subvector_central(v, l, c, f)
      // Se calcula cual de los tres subvectores tiene mayor suma
      // Todas las operaciones son de coste cte
      casos
         si (izq.suma >= der.suma)  -> si (izq.suma >= cent.suma) entonces
                                          optimo := izq
                                       si_no
                                          optimo := cent
                                       fsi
         si (der.suma >= cent.suma) -> optimo := der
         si_no                      -> optimo := cent
      fcasos
   fsi

   devolver optimo
ffun
```

```
fun subvector_central (v:vector, l:ent, c:ent, f:ent) dev vector
  // variables locales
  m, suma, suma_max : ent
  inicio, final : ent // variables que indican el inicio y el final del subvector pasando por el centro
  optimo : vector     // valor devuelto por el metodo

  m := (c+f) / 2      // Se calcula el punto medio del vector
  suma_max := INT_MIN /* Se le asigna el mínimo valor de los enteros para poder hacer las comparaciones
                         con suma, ya que los elementos del vector no valdrán nunca el mínimo */
  suma := 0
  inicio := m - l + 1
  final := m + l - 1  // operaciones con coste cte
  si (inicio < 0) entonces
        inicio := 0   // Se inicializa a cero para que no busque en posiciones inexistentes en el vector
  fsi
  si (final > n-1) entonces
     /* Si se le pasamos una longitud l más grande que la mitad del vector, tomando como posición de
        inicio m, nos pasaremos del rango del vector original. De ahí la asignación final = m + l - 1
        que nos da el límite hasta donde podríamos calcular un subvector de longitud l con inicio en
        el punto medio m. Si final se pasa del rango sólo podremos calcular n - l + 1 subvectores de
        longitud l, o lo que es lo mismo, sólo podremos calcular subvectores con inicio entre 0 y n - l */
     // Calculamos el subvector máximo de forma iterativa
```

```
    para i:=inicio hasta n-l hacer    // se hará n-l-inicio+1 veces
       suma := 0
       para j:=0 hasta l-1 hacer      // se hará l veces
          suma := suma + v[j+i]
       fpara
       si (suma >= suma_max) entonces
          suma_max := suma
          optimo.suma := suma_max
          optimo.ini := i
          optimo.fin := optimo.ini + l - 1
       fsi
    fpara
  si_no
    /* final cae dentro del rango del vector y por tanto se pueden calcular todos los
       posibles subvectores de longitud l que pasen por el centro (m), incluidos los que
       tienen como inicio la mitad del vector (m)*/
    // igual que en el otro caso, se calcula el subvector máximo de forma iterativa
    para i:=inicio hasta m hacer      // m-inicio+1 veces
       suma := 0
       para j:=0 hasta l-1 hacer      // l veces
          suma := suma + v[j+i]
       fpara
       si (suma >= suma_max entonces
          suma_max := suma
          optimo.suma := suma_max     // operaciones con coste cte
          optimo.ini := i
          optimo.fin := optimo.ini + l - 1
       fsi
    fpara
  fsi

  devolver optimo
ffun
```

Este ejemplo de código también está disponible en C.

Organización de un campeonato

El problema de la organización del calendario de un campeonato con Divide y Vencerás.

Multiplicación de enteros grandes

- **Enunciado del problema**: las operaciones aritméticas entre números enteros suelen considerarse operaciones elementales siempre y cuando los números que intervienen no superen los límites de representación. Cuando esto ocurre, hay que acudir a otro tipo de representación, por ejemplo los arrays, y utilizar los algoritmos clásicos de las operaciones aritméticas. El problema planteado consiste en saber si existe alguna forma de reducir el coste en el caso de la multiplicación, ya que el algoritmo clásico de multiplicación es de orden cuadrático. Como alternativa, se podría utilizar el esquema de multiplicación por duplicación, pero no ofrece mejoras en cuanto a tiempo de ejecución.

- **Solución**: considérense dos números enteros, u y v, de n dígitos cada uno.

  1. **Algoritmo básico**: la idea a seguir es descomponer los dos números en la suma de otros dos. Más concretamente, sea S la parte entera de n/2, la descomposición consiste en dividir los números entre 10^S:

     $u \rightarrow w$ representa el cociente y x el resto $\rightarrow$ $u = 10^S \cdot w + x$

     $v \rightarrow y$ representa el cociente y z el resto $\rightarrow$ $v = 10^S \cdot y + z$

  Luego:

     $u \cdot v = (10^S \cdot w + x) \cdot (10^S \cdot y + z) = 10^{2S} \cdot w \cdot y + 10^S \cdot (w \cdot z + x \cdot y) + x \cdot z$

  A continuación, se calculan las suboperaciones $w \cdot y$, $w \cdot z$, $x \cdot y$ y $x \cdot z$. Si los operandos (w, x, y, z) son:

     - "pequeños", entonces se multiplican de la forma clásica.

     - "suficientemente grandes", se aplica de nuevo el algoritmo y se calculan recursivamente las multiplicaciones de los respectivos sumandos.

  2. **Algoritmo mejorado**: al realizar cuatro multiplicaciones de tamaño mitad en cada llamada, el coste resultante es $T(n) = 4 \cdot T(n/2) + g(n)$, donde g(n) es el tiempo invertido en sumas, desplazamientos y operaciones adicionales sencillas, por lo que el algoritmo sigue siendo de orden cuadrático.

  Dado que se necesitan realizar cuatro multiplicaciones de tamaño mitad, no se mejora el tiempo de ejecución respecto a los algoritmos clásicos. Para intentar reducirlo, es preciso evitar alguna de esas multiplicaciones. La clave de la mejora consiste en advertir que no es necesario calcular $w \cdot z$ y $x \cdot y$, si no la suma de ambos. Así, se considera $r = (w + x) \cdot (y + z) = w \cdot y + (w \cdot z + x \cdot y) + x \cdot z$, con los que las multiplicaciones a realizar son:

     $p = w \cdot y$

     $q = x \cdot z$

     $r = (w + x) \cdot (y + z)$

  , hallando la suma de la siguiente forma:

     $(w \cdot z + x \cdot y) = r - p - q$

  Así, volviendo al producto anterior:

     $u \cdot v = (10S \cdot w + x) \cdot (10S \cdot y + z) = 102S \cdot w \cdot y + 10S \cdot (w \cdot z + x \cdot y) + x \cdot z = 102S \cdot p + 10S \cdot (r - p - q) + q$

  , teniendo que calcular las multiplicaciones p, q y r. Si sus operandos (w, x, y, z) son:

     - "pequeños", entonces se multiplican de la forma clásica.

     - "suficientemente grandes" se aplica de nuevo la descomposición anterior, y calculamos recursivamente las multiplicaciones de los respectivos sumandos.

- **Algoritmo básico**:

```
fun Mult (u, v : entero_grande) dev entero_grande;
    var
        n, s : ent;
        w, x, y, z : entero_grande;
    fvar

    n := Max(tamaño(u), tamaño(v));

    si n es pequeño entonces
        dev (MultClásica(u,v));
    sino
        s := n div 2;
        w := u div 10^S;
```

```
        x := u mod 10^S;
        y := v div 10^S;
        z := v mod 10^S;
    fsi

    dev [Mult(w, y)·10^2S + (Mult(w, z) + Mult(x, y))·10^S + Mult(x, z)];
ffun
```

- **Algoritmo mejorado**:

```
fun Mult2 (u,v : entero_grande) dev entero_grande;
    var
        n, s : ent;
        w, x, y, z : entero_grande;
    fvar

    n := Max(tamaño(u),tamaño(v));

    si n es pequeño entonces
        dev (MultClásica(u,v));
    sino
        s := n div 2;
        w := u div 10^S;
        x := u mod 10^S;
        y := v div 10^S;
        z := v mod 10^S;
        r := Mult2(w + x,y + z);
        p := Mult2(w,y);
        q := Mult2(x,z);
    fsi

    dev [p·10^2S + (r - p - q)·10^S + q];
ffun
```

Con lo cual, el coste final e: $T(n) = 3 \cdot T(n/2) + g(n)$, perteneciente al orden $\Theta(n^{1,59})$.

- **Otras consideraciones**:
  - Si *u* y *v* son de distinta longitud (*m* y *n* respectivamente, con $m < n$), entonces:

    - si no difieren en más de un factor de 2, se rellena el operando más pequeño con ceros no significativos e igualar las longitudes.

    - si uno de los operandos es mucho mayor que el otro, se segmenta el operando más largo, *v*, en bloques de tamaño *m* y se utiliza el algoritmo para multiplicar cada uno de los bloques de *v* por *u*. De esta forma, se multiplican operandos de igual tamaño *m*. Finalmente, el producto definitivo, se halla con adiciones y desplazamientos sencillos.
  - Los números de longitud impar se multiplican de la misma forma que los pares. Para ello, se dividen del modo más equitativo posible: un número de *n* cifras (*n* impar), se divide en uno más significativo |n/2| cifras y otro menos significativo de |n/2|+1 cifras.

Búsqueda binaria

```
algoritmo BusquedaBinaria(a: vector; prim,ult,x:enteros)
var tercio:entero;
inicio
   si (prim>=ultimo) entonces return a[ult]=x;
   sino
      tercio:=prim+((ult-prim+1)DIV 3);
      si x=a[tercio] entonces return true;
      sinosi (x<a[tercio] entonces return BusquedaBinaria(a,prim,tercio,x);
      sino return BusquedaBinaria(a,tercio+1,ult,x)
      finsi
   finsi
 fin BusquedaBinaria;
```

Fuentes y contribuyentes del artículo

Algoritmia/Divide y vencerás *Fuente*: http://es.wikibooks.org/w/index.php?oldid=170575 *Contribuyentes*: Akhram, Der Künstler, Gothmog, Oleinad, 7 ediciones anónimas

Licencia