DIVIDE Y CONQUISTA

El Esquema de Divide y Conquista es una aplicación directa de las técnicas de diseño recursivas de algoritmos.
Hay dos rasgos fundamentales que caracterizan los problemas que son resolubles aplicando este esquema:

- Es necesario que el problema admita una formulación recursiva. Se debe resolver el problema inicial a base de combinar los resultados obtenidos en la resolución de un número de subproblemas (reducido). Estos son del mismo tipo que el problema inicial pero han de trabajar con datos de tamaño estrictamente menor
- El tamaño de los datos que manipulan los subproblemas ha de ser lo mas parecido posible. Si n es el tamaño del problema inicial entonces n/c siendo c>0 una constante natural, denota el tamaño de los datos que recibe cada uno de los subproblemas en que se descompone.

En este tipo de algoritmos se realizan operaciones para ***fragmentar*** el tamaño de los datos y para ***combinar*** los resultados de los diferentes subproblemas resueltos.
Ahora bien, para que sea conveniente en términos de eficiencia aplicar esta estrategia de resolución, se deben dar condiciones adicionales como:

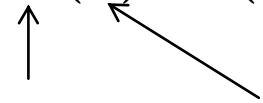
- Nunca se resuelve el mismo problema mas de una vez
- Las operaciones de fragmentar y combinar deben ser eficientes
- El tamaño de los subproblemas lo mas parecido posible
- Evitar generar nuevas llamadas cuando el tamaño de los datos que recibe el subproblema es suficientemente pequeño y allí utilizar el algoritmo del caso directo.

La abstracción de control para un algoritmo de este tipo es:

```
Type DyC (P)
   {if Small(P)  ──────────────────►  determina si el tamaño de la entrada es
        return S(P);                  lo sufic. Chico que no necesita ser particionado
     else
    {Divide P en P1, P2, ....Pk;   k>1
     Aplicar DyC a c/Pi
     return Combina (DyC(P1), DyC(P2)....,DyC(Pk));
   }
}
```

El costo del algoritmo genérico de DyC se puede expresar de la siguiente manera:

T(n) =k. T(n/c) + costo (max(Divide, Combina))

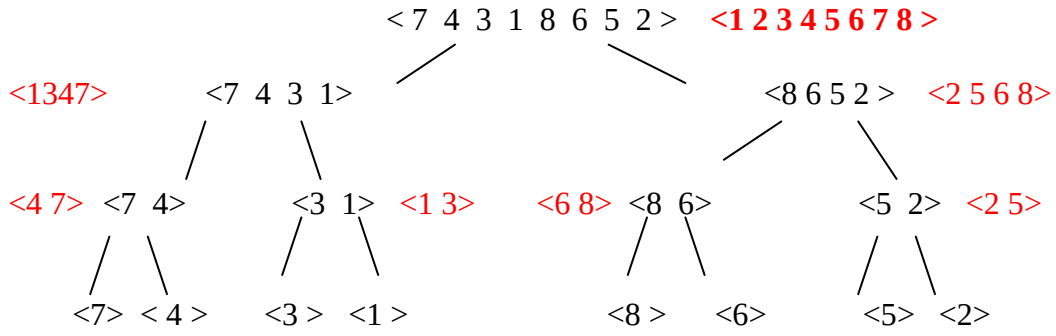


k fragmentos tamaño de los subproblemas

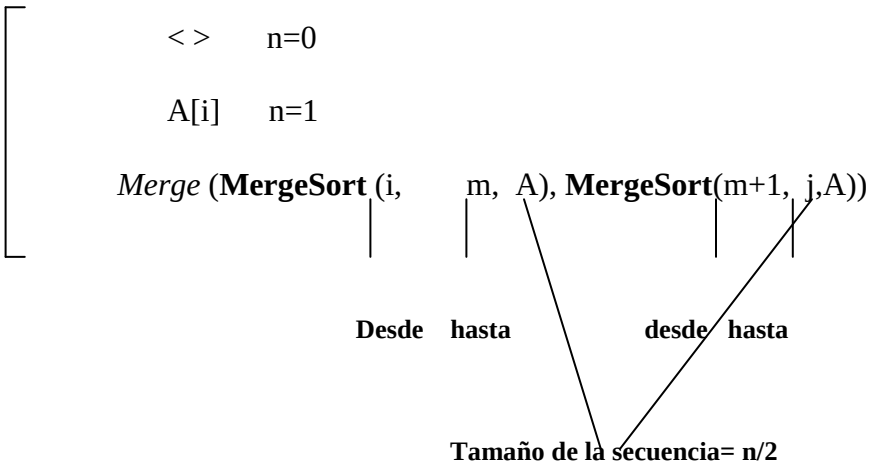
Vamos a ver dos problemas (de sorting), en los cuales se utiliza este esquema para su resolución y ver las situaciones posibles que pueden darse en ellos.

Algoritmo de ordenamiento: MergeSort:
Supongamos que se ha de ordenar una secuencia de n elementos. La solución Divide y Conquista aplicada consiste en fragmentar la secuencia de entrada en 2 subsecuencias de tamaño similar, ordenar recursivamente cada una de ellas, y finalmente fusionarlas de modo adecuado para generar una sola secuencia en la que aparezcan todos los elementos ordenados.

Supongamos la siguiente secuencia:



La forma recurrente para este problema es:



Divido el problema en 2 subsecuencias de n/2 elementos cada una, hasta que tiene longitud uno, en cuyo caso no hay mas trabajo para hacer (ya está ordenada).

La operación clave del MS es el merging de las dos secuencias ordenadas en un paso. Esta corresponde a la función genérica de combinar. Esta función supone que las subsecuencias están ordenadas y las ´junta´ para formar una subsec. ordenada.

```
Void MergeSort (int liminf, int limsup)
// a[liminf:limsup] arreglo global
{if (liminf<limsup)
   //divido en subproblemas
     int medio=(liminf+limsup)/2;
   // resuelvo los subproblemas
    MergeSort (liminf, medio);
    MergeSort (medio+1, limsup);
     // combino las soluciones
     Merge (liminf, medio,limsup);
  }
```

Para realizar el Merge, razonamos de la siguiente forma: tenemos un arreglo global conteniendo dos subconjuntos ordenados en a[liminf:medio] y en a[medio+1:limsup]. El objetivo es fusionar estos dos conjuntos en uno que reside en a[liminf:limsup]. b[] es un arreglo auxiliar global.

Mostrar transparencia Función Merge

Podemos calcular el costo para el algoritmo del MergeSort y lo planteamos de la siguiente manera:

$$T(n) \begin{cases} c & \text{cuando } n=1 \\ 2\,T(n/2) + c1\ n + c2 & \end{cases}$$

$2\,T(n/2)$: complejidad del Merge; siendo $n=2^k$

$T(n) \in O(n \log_2 n)$

Mostrar transparencia cálculo de T(n)

Otro ejemplo: Algoritmo de Ordenamiento QuickSort

El enfoque de DyC puede ser usado para lograr otro método de ordenamiento eficiente diferente al MergeSort. Este es el QuickSort, donde también se aplica el principio de DyC pero a diferencia del anterior no garantiza que cada subejemplar tendrá un tamaño que es fracción del tamaño original.

La base del QuickSort es el procedimiento de partición. Este procedimiento sitúa al pivote en su lugar apropiado. Luego no queda mas que ordenar los segmentos que quedan a su izquierda y a su derecha. Mientras que en MergeSort la división es simple y el trabajo se realiza durante la fase de combinación, en QuickSort sucede lo contrario.

En el QS la división del conjunto de elementos es hecha de manera tal que los subconjuntos o subarreglos no necesitan ser ordenados luego. Esto se logra por reacomodamiento de los elementos en a[1:n] tal que a[i] <=a[j] para todo i entre 1 y m y para todo j entre m+1 y n para algún **m, 1<=m<=n.** Así los elementos en a[1:m] y a[m+1:n] pueden ser ordenados independientemente. El reacomodamiento de los elementos es logrado tomando algún elemento de a[], el pivote **t**=a[s] y luego reordenando los otros elementos, por eso todos los elementos que aparecen después de t son >= a **t.** Esto es la partición.

Abstracción DyC del QuickSort

```
QuickSort (int a[], int l , int r)
    {int i;
     if (r>l)
       { i=particion (l,r);
     // i es la posición del pivote //
         QuickSort (a,l,i-1);
         QuickSort (a,i+1,r);
       }
     }
```

l y r delimitan las particiones. Las condiciones que deben valer son:

1) el elemento a[i] está en su lugar final en el arreglo para cierto i.
2) Todos los elementos en a[1]...a[i-1] son <= que a[i]
3) Todos los elementos en a[i+1]..a[r] son >= que a[i]

Estrategia:

a) Elegir arbitrariamente a[r] a ser el elemento que estará en su posición final.
b) Recorrer desde el límite izquierdo del arreglo hasta que un elemento sea >= que a[r], recorrer desde el límite derecho del arreglo hasta que un elemento sea <= que a[r].
c) Estos dos elementos que encontramos están fuera de lugar, por lo tanto se intercambian.

Continuando de esta manera, aseguramos que todos los elementos a la izquierda del punto izquierdo son menores que a[r] y todos los elementos del arreglo a la derecha son mayores que a[r].

Cuando los punteros se cruzan el proceso de partición está casi completo; lo que falta es intercambiar a[r] con el elemento de mas a la izquierda de la parte derecha.

QS presenta un comportamiento menos homogéneo que por ej. MS, en él el tamaño de los fragmentos a ordenar depende de la calidad del pivote. Un buen pivote, por ejemplo la mediana construye dos fragmentos de tamaño n/2, pero un mal pivote puede producir un

fragmento de tamaño 1 y otro de tamaño n-1. En el caso de utilizar uno bueno la recurrencia es $T(n)=2T(n/2)+O(n)$ → lo que da un costo de O(nlogn). En el otro caso la recurrencia es absolutamente diferente $T(n)=T(n-1) + O(n)$ → $O(n^2)$. Estos son los costos en el caso mejor y en el peor, a pesar del costo en el peor caso su comportamiento promedio es O(n log n) lo que lo convierte en un algoritmo eficiente.

QuickSort completo

```
QuickSort (int a[], int l , int r)
    {int v,i,j,t;
     if (r>l)
       {v=a[r]; i=l-1; j=r;
       { i=particion (l,r);
     // i es la posición del pivote //
          QuickSort (a,l,i-1);
          QuickSort (a,i+1,r);
        }
   }
    if i<j
      exchange a[i] <-->a[j]
     else
      return j
```

O(n) ya que n=r-p+1

Un ejemplo del seguimiento del QS:

A S O R T I N G E X A M P L E

Elegimos E de mas a la derecha como elemento para determinar la partición.

1) El rastreo desde la izquierda *para* en S; el rastreo desde la derecha *para* en A.

A S .....*idem*..... A M P L E

Luego, se realiza el intercambio
A A .....S M P L E

2) Luego el rastreo desde la izquierda *para* en la O y el de la derecha *para* en la E.

A A O .....E X S M P L E

Luego, se realiza el intercambio

A A E ..... O X M P L E

Continuando con el ciclo de buscar uno mayor por la izquierda y menor por la derecha termino en la siguiente situación

A A E R T I N G O X S M P L E

Aquí significa que tengo que cruzar los punteros:

A A E **E** T I N G O X S M P L R

Y queda particionado

FUNCION MERGE

```
Void Merge (int liminf, int medio, int limsup)
   {int h=liminf, i=liminf, j=medio+1, k;
    while ((h<=medio) && (j<=limsup))
    {if (a[h]<=a[j])
     {b[i]=a[h];
        h++;
      }
     else
      {b[i]=a[j];
        j++;
      }
      i++;
     }
    if(h>medio) for (k=j;k<=medio;k++) {
              b[i]=a[k]; i++;
             }
     else for (k=h;k<=medio;k++) {b[i]=a[k]; i++;
             }

       for (k=liminf;k>=limsup;k++)   a[k]=b[k];
}
```

Control de quien coloco en b

Termino de pasar los elementos a b

Copio b en a

El costo de la función Merge es O(n)

COSTO COMPUTACIONAL DEL MERGESORT

$T(n) = 2\,T(2^{k-1}) + c1\,2^k + c2$

$2[2T(2^{k-2}) + c1\,2^{k-1} + c2] + c12^k + c2$

$2^2\,T(2^{k-2}) + 2\,c12^{k-1} + 2\,c2 + c12^k + c2$

$2^2\,[2T(2^{k-3}) + c1\,2^{k-2} + c2] + 2\,c12^{k-1} + 2\,c2 + c12^k + c2$

$2^3\,T(2^{k-3}) + c12^k + 2^2\,c2 + 2\,c1\,2^{k-1} + 2\,c2 + c1\,2^k + c2$

$$2^3\,T(2^{k-3}) + 3\,c12^k + \sum_{j=0}^{2} 2^j$$

$$2^i\,T(2^{k-i}) + i\,c12^k + \sum_{j=0}^{i-1} 2^j$$

cuando $i=k \rightarrow T(1)$

$2^k\,c + k\,2^k\,c1 + c2\,(2^k - 1)$

ya que $n=2^k$

$2^{\log_2 n} + n \log_2 n\,c1 + c2\,(2^{\log_2 n} - 1)$

$T(n) \in O(n \log_2 n)$