La estrategia Divide y Vencerás:

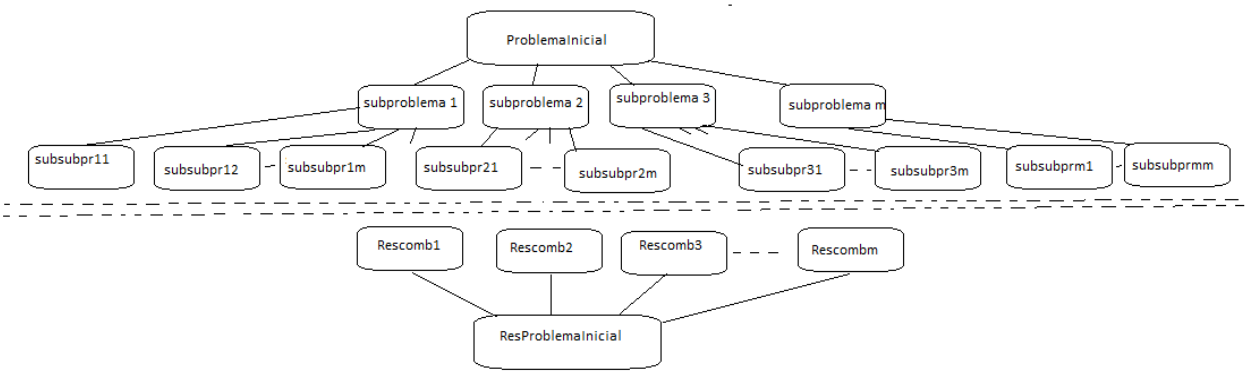
Es frecuente indicar, en diversas disciplinas, que se aplica una resolución Divide y Vencerás cuando se descompone un problema complejo en partes mas simples para tratarlas por separado y facilitar así el desarrollo de la solución.

La idea de dividir para conquistar se precisa de forma mucho mas rigurosa en el contexto de las llamadas estrategias de resolución de problemas, Divide y Vencerás (DyV) de Matemática o Informática. Se trata de una forma de construir o diseñar algoritmos según etapas establecidas y puede ser enunciada de la siguiente manera:

Si el tamaño del problema en cuestión es N>L, siendo L un valor arbitrario, se debe

- dividir el problema en m subproblemas independientes (es decir, no solapados), de la misma estructura que el problema original (es decir que estaremos resolviendo, en cada uno de esos m subproblemas, lo mismo que en el problema original). Si los subproblemas tienen tamaño similar entre si, es mejor.
- Para cada uno de los m subproblemas, procedemos del mismo modo que con el problema original, considerando su tamaño, y dividiéndolo nuevamente en m subproblemas independientes, o no.

Si, en cambio, el tamaño del problema no supera el valor de L, lo resolvemos de otra forma, utilizando generalmente un camino más directo.
Posteriormente, la solución del problema original se obtiene por combinación de las m soluciones obtenidas de los subproblemas en que se lo había dividido.



Como puede inferirse, hay una cercanía conceptual entre la idea de la estrategia Divide y Vencerás (o Dividir y Conquistar, "Divide and Conquer') y los algoritmos recursivos, dado que la división del problema en m subproblemas de la misma estructura sugiere llamados recursivos en donde se ha reducido el tamaño de la entrada de la forma indicada, y la solución que no aplica la estrategia, cuando el tamaño del problema no supera a L se correspondería con el caso base. Pero no hay que pensar que la estrategia implica necesariamentea aplicar recursividad. Estos algoritmos podrían ser implementados usando formas no recursivas, por ejemplo con pilas para almacenar los resultados parciales y otros parámetros. Recordar que, como demuestra el teorema fundamental de las Ciencias de la Computación, todo algoritmo recursivo puede reescribirse iterativamente, y viceversa.

Por otro lado, al describir la idea de DyV se ha hecho mención al 'tamaño del problema'. ¿A qué nos referimos con esto? En general, cuando resolvemos un problema computacional podemos indicar qué elemento o elementos tienen responsabilidad directa en el consumo de recursos del algoritmo, en particular del recurso espacio (de memoria) y tiempo (de ejecución). En general, es el número de datos sobre los cuales va a trabajar el algoritmo (por ejemplo, si se trata de hacer un ordenamiento en un array), o el valor del dato en si (por ejemplo, si se trata de calcular el factorial de un número

entero positivo). Hilando un poco más fino, se puede definir el tamaño de la entrada de un problema como el número de símbolos necesarios (de un cierto alfabeto, finito, obviamente) necesarios para poder codificar todos los datos de un problema.

Entre los algoritmos más conocidos que utilizan DyV exitosamente están: la búsqueda binaria (o dicotómica) en un array ordenado, la potencia entera de enteros, los ordenamientos internos Mergesort, Quicksort, Radixsort, la resolución de las Torres de Hanoi, la multiplicación de enteros grandes de Karatsuba y Ofman, la multiplicación de matrices de Strassen, el cálculo de la Transformada rápida de Fourier, la determinación del par de puntos más cercanos en un plano, el armado de un fixture para un campeonato. Algunos de ellos se desarrollarán a continuación.

Existen también problemas que, si bien pueden ser resueltos aplicando DyV, no solo no presentan mejoras de eficiencia al utilizar la estrategia, sino que más aún, resultan más costosos en terminos temporales que su resolución sin aplicarla. Tal es el caso del problema de la subsecuencia de suma maxima en un array (se propone determiner cual es la suma maxima que puede obtenerse considerando elementos contiguos contenidos en un array), problema que puede resolverse con un O(n log n) para el peor caso por D y V , y con un O(n) con una exploración lineal.

Ejemplo 1: mergesort

Uno de los problemas que más claramente muestra la aplicación de la estrategia DyV es el ordenamiento por Mezcla, o MergeSort.

Este ordenamiento plantea lo siguiente:
Para ordenar un array de N elementos (tamaño N) proceder así: si N es mayor que 1, dividir el array a la mitad, ordenar cada mitad e intercalar ordenadamente las mitades ordenadas. Si no, el array ya está ordenado.

```
void mergesort (double v[], int pr, int tul )
{
 if (pr < ul)
   {
    int me= (pr+ul)/2;
    mergesort (v, pr, me);
    mergesort(v, me+1, ul);
    intercalar (v, pr, me, ul);     // este proceso realiza la intercalación ordenada de las mitades ordenadas
   }
}
```

Aquí vemos lo siguiente: N (dado por ul – pr +1) es el tamaño del problema. El valor arbitrario con el cual se compara ese tamaño es 1. Si el array a analizar tiene 1 o menos elementos, puede considerárselo ordenado. Si no, lo partiremos en dos (cada uno de los subarrays corresponde a un subproblema en el cual tenemos que resolver la misma situación que en el problema original, es decir, un ordenamiento). Observar que los dos problemas son además, independientes, porque, obviamente, no hay zonas superpuestas entre el primer subarray y el segundo.
La combinación de soluciones parciales que nos lleva a la solución del problema original es en este caso la intercalación ordenada de las mitades ordenadas del vector, en cada etapa del array.

Análisis complejidad temporal:

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

T(1) =1

Por aplicación del Teorema Maestro de Reducción por División resulta T(n) pertenece a O (n * log n)

En esta version de mergesort la cantidad de subproblemas es 2, dado que partimos al array a la mitad (o casi), y, como hemos visto antes, L, el valor arbitrario que establece el umbral a partir del cual volver a aplicar DyV es 1.

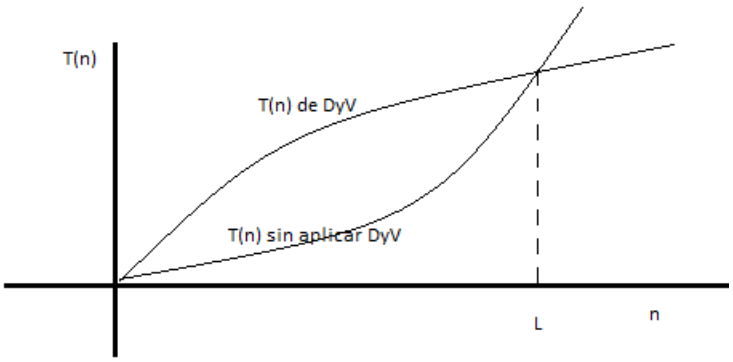
Ahora bien ¿debe ser asi siempre? ¿no puede plantearse un valor diferente para el corte? ¿No podríamos, por ejemplo, partir el array en 3, 4 , o más partes y establecer que se aplicará DyV cuando el tamaño N supere 5, 10, 12, o lo que nos parezca?

Podemos plantear versiones diferentes del Mergesort. Sólo habrá que adaptar convenientemente la intercalación ordenada, como se muestra a continuación:

```
void msrecharged (double v[], int pr, int ul)
{
  if ((ul-pr) > 10)
    {
      int ter1= (pr+ul) / 3;
      int ter2=ter1 *2;
      msrecharged (v, pr,ter1);
      msrecharged(v,ter1+1, ter2);
      msrecharged(v, ter2+1, ul);
      intercalar (v, pr,ter1, ter2, ul);
   }
else bubblesort (v, pr, ul);
}
```

Pero, en general, ¿cual será el umbral (valor de L) más adecuado? ¿Cual será el que nos permita obtener los resultados con la mayor rapidez? Por ejemplo, si estamos partiendo el array a la mitad, será necesario plantear que partiremos mientras N>L o lo mejor sería cortar antes? Y en qué punto habría que hacer el corte? Hay que considerar que un algoritmo desarrollado con la estrategia DyV podría tener un T(n) que superara el correspondiente al algoritmo sin aplicar la estrategia para valores reducidos de n.
La determinación del mejor L no es un problema simple. Puede estimarse básicamente de dos formas:

- Empíricamente: construyendo tablas para ambos algoritmos (aplicando DyV y sin aplicarlo) para valores de n crecientes y estableciendo con ellas el tamaño de la entrada para el cual se igualan los valores en ambas tablas.
- Teóricamente: si tenemos las expresiones formales correspondientes a T(n) para cada algoritmo, matemáticamente se calcula el *n* para el cual la solución aplicando D y V y la solución que no aplica esa estrategia tienen igual coste.



Ejemplo 2: quicksort

El Ordenamiento rápido, o Quick Sort en cada etapa ubica en su posición definitiva un elemento distinguido llamado pivote, dejando (si estamos ordenando de forma creciente) los menores o iguales que él a la izquierda y los mayores a derecha. Las zonas derechas e izquierda del pivote, eventualmente desordenadas, serán ordenadas en etapas posteriores siguiendo el mismo esquema algorítmico.

El proceso de ubicación del pivote constituye aca la división del problema en dos subproblemas. Los subproblemas serán las zonas izquierda y derecha del pivote una vez ubicado éste, ambos obviamente de la misma estructura que el original. La combinación de la soluciones para llegar a la resolución del problema inicial es un paso trivial.

Lo que se analizará cuando veamos complejidad algorítmica es la influencia del desbalanceo de tamaños entre el subvector izquierdo y el derecho en el coste temporal del algoritmo.

//versión recursiva del ordenamiento rápido.

```
void quicksort (double v[], int pr, int ul)
{
  if ( ul >pr)
    {
      int pospiv = Ubicarpivote (v, pr, ul);
      quicksort (v, pr, pospiv-1);
      quicksort(v, pospiv+1, ul );
    }
}
int Ubicarpivote(int v[] int pr, int ul) {
    int ppiv= pr;
    int k= pr;
    int j= k+1;
    while (j<=ul) {
      if (a[j]<a[ppiv] ) {
        k= k+1;
        intercambiar(a, k, j);
      }
      j= j+1;
    }
    intercambiar(a, k, ppiv);
    return k;
  }
```

Análisis de complejidad temporal:

Tenemos que considerar que al ubicar el pivote, los subvectores que quedan al lado izquierdo y derecho del pivote pueden tener tamaños parecidos o muy disímiles.

Esto nos lleva a considerar dos 'situaciones límite', a saber:

1. que cada vez que ubiquemos el pivote el subvector lado izquierdo del pivote tenga el mismo tamaño (o casi) que el subvector del lado derecho del pivote,

2. o bien, que cada vez que ubiquemos el pivote, uno de los dos subvectores quede sin elementos y el otro tiene n-1 elementos

Si en cada etapa se da la situación 1, entonces las ecuaciones que resultan son:

T (n) = 2 * T(n/2) + n

T (1) = 1

Que, como ya hemos visto, nos llevan a que T(n) pertenece a O (n * log n).

Si, en cambio, en cada etapa se da la situación 2, entonces las ecuaciones que resultan son:

T (n) = T(n - 1) + n

T (0) = 1

Aquí no se puede aplicar el Teorema Maestro de reducción por división. Resolviendo por otro método llegamos a que T(n) pertenece a O (n^2).

Esto nos dice que la versión que planteamos del ordenamiento rápido, en el peor caso tiene un coste temporal cuadrático (aunque con baja probabilidad).

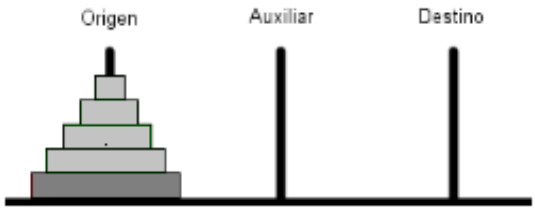
Existen versiones del ordenamiento rápido que introducen modificaciones para reducir drásticamente las probabilidades de que el coste sea tan alto.

Una buena estrategia para elegir el pivote, puesto que estadísticamente se prueba que reduce la posibilidad de desbalanceo entre los tamaños de los subarrays izquierdo y derecho del pivote es trabajar con la mediana de tres elementos: de la izquierda, la derecha y el centro. Es decir que, si el array de datos es A, se tomará la mediana entre A[0], A[N-1] y A[(N-1)/2] . A continuación, el código de esta versión de quicksort (recursivo).

```
quicksort (int A[], int izq, int der)
{
if (der - izq > 1)
 {
  int i = ubicarpivote (A, izq, der);
  quicksort (A, izq, i - 1);
  quicksort (A, i + 1, der);
  }
}

 int ubicapivote (int A[], int izq, int der)

// se elegirá el pivote y se colocará en A[der-1]

  int centro = div2 (izq + der);
  if   (A[izq] > A[centro])
       {intercambiar (A, izq, centro);}
  if (A[izq] > A[der])
       {intercambiar (A, izq, der);}
  if (A[centro] > A[der]) {
```

```
        {intercambiar (A, centro, der);}
  intercambiar (A, centro, der - 1);
 }

// se particiona el array
  int i = izq, j = der - 1;
  int pivote = A[der - 1];
  do
    {
     do
         i++;
    while (A[i] < pivote);
    do
          j--;
   while (A[j] > pivote);
   intercambiar (A, i, j);
}
while (j > i);
// se elimina el último intercambio porque se realizó sin cumplirse i<j
intercambiar (A, i, j);
//se coloca el pivote en la posición definitiva
intercambiar (A, i, der - 1);
}
```

Ejemplo 3: las Torres de Hanoi:



Es un juego que consta de tres varillas, *origen, destino* y *auxiliar* y una serie de discos de distintos tamaños, colocados de mayor a menor tamaño en la columna origen. El juego consiste en pasar todos los discos de la columna origen a la columna destino usando la columna auxiliar, moviéndolos de a uno y manteniéndolos siempre ordenados de mayor a menor.

El problema se puede resolver de forma muy sencilla usando recursividad y la técnica divide y vencerás. Si hay un solo disco (caso base), entonces se lo lleva directamente de la varilla *origen* a la varilla *destino*. Si hay más (n>1 es el caso general) entonces:

Se llevan n-1 discos de la varilla *origen* a la *auxiliar*.

Se lleva un solo disco (el que queda) de la varilla *origen* a la *destino*

Se traen los n-1 discos de la varilla *auxiliar* a la *destino*. Es decir:

void Hanoi (int n, char o, char d, char a)

```
{
  if (n>0)
    {
      Hanoi (n-1, o, a, d);
      std::cout<< "mover un disco desde" <<o <<"hasta"<<d<<endl;
      Hanoi (n-1, a, d, o);
    }
}
```

Análisis de complejidad temporal:

$T(n) = 2* T (n-1) +1$

$T(0)=1$

$T(n-1)= 2* T(n-2) +1$

Entonces

$T(n)=2(2* T(n-2) +1) +1$

$T(n)= 4 T(n-2) +2 +1$

Como es

$T(n-2)=2* T(n-3) +1$

Entonces

$T(n)=4(2* T(n-3)+1)+2+1$

$T(n)=8* T(n – 3) +4 + 2 + 1$

En el iésimo paso

$T(n) = 2^i * T (n – i) + \sum_0 \quad 2^j$

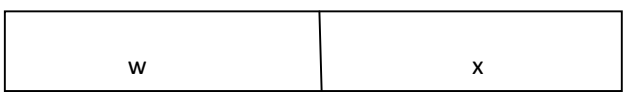
Lo cual lleva a T(n) pertenece a $O(2^n)$

Ejemplo 4: multiplicación de Enteros Grandes (A. Karatsuba e Y. Ofman)

Consideremos que se debe multiplicar dos números enteros de muchas cifras. Son tan extensos que la cantidad de cifras determinará el coste temporal. Los enteros serán u y v y tendrán n cifras. Si uno de ellos tienen menos, consideraremos que se completa con ceros a izquierda.

El planteo obviamente es independiente de la base elegida, si bien lo desarrollaremos en base 10. Podemos considerer a ambos numero divididos en dos 'tramos'de igual tamaño (que nos van a permitir plantear subproblemas):

u



v

En símbolos,

$u = 10^s w + x$

$v = 10^s y + z$

con $0 <= x < 10^s$ $0 <= z < 10^s$ $s = [n/2]$ (tener en cuenta que s es **piso** de n/2)

Resulta:

$u * v = 10^{2s} w y + 10^s (wz + xy) + xz$

Esta expresión indica como debe realizarse la partición en subproblemas.

De este modo, el esbozo del primer código de multiplicación quedaría así:

```
Enterogrande mul (enterogrande u, enterogrande v)
{
  int  n = max(u.tamanyo(), v.tamanyo();   //como medimos los tamaños??
  if (n >L)        // en que valor convendrá fijar L??
    {
      int  s= n/2 ;
      Enterogrande w= u/ 10^s ;
      Enterogrande x=u % 10^s ;
      Enterogrande y= v/10^s ;
      Enterogrande z=v%10^s ;
      return mul (w,y ) * 10^2s + (mul (w,z) + mul (x,y)) * 10^s + mul (x, z);
   }
 else return u * v;
}
```

¿Este algoritmo será más eficiente que el que no aplica DyV? Recordar siempre que nos interesa aplicar una estrategia específica si esta estrategia representa una ventaja respecto a otras alternativas (si hay otras alternativas).
Análisis de complejidad temporal de la Multiplicación de enteros grandes (recursiva) aplicando Divide y Vencerás sin los reemplazos de Karatsuba:

Las sumas, multiplicaciones por potencias de 10 y divisiones por potencias de 10 tienen un coste que es lineal con respecto a la entrada.La multiplicación de dos enteros de tamaño n tiene un coste O(n).

Entonces, si contamos cuántas operaciones y de que tipo se realizan, y cuántas invocaciones recursivas hay en mul, se tiene:

$T(n) = 4 * T(n/2) + O(n)$ y $T(1)=1$

Observar que, puesto que cada número se reduce a la mitad en las invocaciones, el tamaño indicado para la entrada debe pasar d en a n/2.

El ultimo término de la primera ecuación indica que hay un grupo de sentencias que son O(n). Estas sentencias corresponden a las sumasy multiplicaciones por potencias de 10.

Entonces, mul tiene un coste O(n). Lo cual nos indica que no hemos ganado nada, con el solo hecho de aplicar la estrategia DyV. En general hay que recordar que una estrategia no representa una panacea en si; la clave está en usarla de una forma adecuada, o suficientemente ingeniosa. Eso es lo que se logra, en este caso, con el reemplazo propuesto por Karatsuba.

Karatsuba propuso calcular wy, wz+xy, y xz haciendo menos de cuatro multiplicaciones:

Dado que r = (w + x)(y + z) = wy + (wz + xy) + xz

Se puede desarrollar esta función:

```
Enterogrande mul2 (enterogrande u, enterogrande v)
{
 int  n = max(u.tamanyo(), v.tamanyo();   //como medimos los tamaños??
 if (n >L)        // en que valor convendrá fijar L??
   {
     int  s= n/2 ;
     Enterogrande w= u/ 10^s ;
     Enterogrande x=u % 10^s ;
     Enterogrande y= v/10^s ;
     Enterogrande z=v%10^s ;
     Enterogrande r= mul2 (w+x, y +z);
     Enterogrande p= mul2 (w , y);
     Enterogrande q = mul2 (x, z);
     return p * 10^2s  + (r-p-q) * 10^s + q;
  }
 else return u * v;
}
```

Al analizar el código de mul2, se puede ver que se reduce el número de invocaciones. Es cierto que hay más sumas y restas, pero estas son operaciones de coste significativamente menor.
Así, resulta, para mul2:

$T(n) = 3\ T\ (n/2) + n \quad T(1) = 1$

Y resulta entonces que T(n) pertenece a $O\ (n^{\log_2 3})$, lo cual muestra la conveniencia del algoritmo para n grande.

Ejemplo 5: algoritmo de Strassen para la multiplicación de matrices

El objetivo de este algoritmo es obtener el producto de matrices cuadradas, de n x n. La suma de matrices de n x n pertenece, como sabemos, a $O(n^2)$, y el producto de matrices cuadradas de n x n pertenece a $O(n^3)$.

Inicialmente plantearemos el producto d edos matrices de n x n. Llamémoslas A y B. Si no fueran cuadradas y de la misma dimensión, podríamos completarlas convenientemente con 0.

A x B = C
Son matrices cuadradas, y realizaremos una partición de cada una en 4 submatrices. Cada submatriz tendrá n/2 filas y columnas.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Siendo

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$
$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$
$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$
$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

y así se continúa dividiendo las submatrices mientras n sea mayor que el umbral propuesto.
De este modo, estamos realizando 8 multiplicaciones y 4 sumas para desarrollar el algoritmo de multiplicación de matrices. (Se deja como ejercicio simple el desarrollo del algoritmo).

¿Se habrá logrado alguna ventaja en el sentido de la eficiencia?

Las ecuaciones asociadas son

$T(n) = 8\ T(n/2) + n$
$T(1) = 1$
Lo cual nos lleva a que T (n) pertenece a $O(n^3)$. Lo que indica que la aplicación de DyV para n grande en estas condiciones no representa ninguna mejora con respecto a la multiplicación de matrices habitual.
Strassen propuso mejoras aplicando DyV pero con estos reemplazos:

m1 = (a21 + a22 - a11)(b22 - b12 + b11)
m2 = a11b11
m3 = a12b21
m4 = (a11 - a21)(b22 - b12)
m5 = (a21 + a22)(b12 - b11)
m6 = (a12 - a21 + a11 - a22)b22
m7 = a22(b11 + b22 - b12 - b21)

Ahora se tienen 24 sumas y restas, pero 7 multiplicaciones; antes eran 8. (Se deja como ejercicio el desarrollo complete y la codificación del algoritmo de multiplicación de matrices cuadradas por Strassen).
Al analizar el código de mul2, se puede ver que se reduce el número de invocaciones. Es cierto que hay más sumas y restas, pero estas son operaciones de coste significativamente menor. Así, resulta:

$T(n) = 3\ T(n/2) + n \quad T(1) = 1$
Y resulta entonces que T(n) pertenece a $O(n^{\log_2 3})$, lo cual demuestra que este algoritmo de multiplicación de matrices para n grande reduce el coste temporal con respecto al algoritmo tradicional.

Ejemplo 6: la transformada rápida de Fourier

El problema de la transformada rápida de Fourier consiste en resolver lo siguiente:

$$A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}, 0 \le j \le n-1$$

donde $i = \sqrt{-1}$ y $a0, a1, \ldots, an\_1$ son números dados.

Dado $w_n = e^{\frac{2\pi i}{n}}$,

$$A_j = a_0 + a_1 w_n^j + a_2 w_n^{2j} + \cdots + a_{n-1} w_n^{(n-1)j}.$$

Sea $n = 4$. Así, se tiene

$$\begin{aligned}
A_0 &= a_0 + a_1 + a_2 + a_3 \\
A_1 &= a_0 + a_1 w_4 + a_2 w_4^2 + a_3 w_4^3 \\
A_2 &= a_0 + a_1 w_4^2 + a_2 w_4^4 + a_3 w_4^6 \\
A_3 &= a_0 + a_1 w_4^3 + a_2 w_4^6 + a_3 w_4^9.
\end{aligned}$$

Y resulta:

$$\begin{aligned}
A_0 &= (a_0 + a_2) + (a_1 + a_3) \\
A_1 &= (a_0 + a_2 w_4^2) + w_4(a_1 + a_3 w_4^2) \\
A_2 &= (a_0 + a_2 w_4^4) + w_4^2(a_1 + a_3 w_4^4) \\
A_3 &= (a_0 + a_2 w_4^6) + w_4^3(a_1 + a_3 w_4^6)
\end{aligned}$$

Dado que $w_n^2 = w_{n/2}$ y $w_n^{n+k} = w_n^k$, es:

$$\begin{aligned}
A_0 &= (a_0 + a_2) + (a_1 + a_3) \\
A_1 &= (a_0 + a_2 w_2) + w_4(a_1 + a_3 w_2) \\
A_2 &= (a_0 + a_2) + w_4^2(a_1 + a_3) \\
A_3 &= (a_0 + a_2 w_4^2) + w_4^3(a_1 + a_3 w_4^2) \\
&= (a_0 + a_2 w_2) + w_4^3(a_1 + a_3 w_2).
\end{aligned}$$

Tomando

$$B_0 = a_0 + a_2$$
$$C_0 = a_1 + a_3$$
$$B_1 = a_0 + a_2 w_2$$
$$C_1 = a_1 + a_3 w_2.$$

Resulta

$$A_0 = B_0 + w_4^0 C_0$$
$$A_1 = B_1 + w_4^1 C_1$$
$$A_2 = B_0 + w_4^2 C_0$$
$$A_3 = B_1 + w_4^3 C_1.$$

Es decir, hay que calcular, B_0, C_0, B_1 y C_1. A partir de A_0 se puede obtener A_2 y a partir de A_1 se puede obtener A_3. Las B_i son las transformadas de Fourier de los datos de entradacon número impar y las C_i las de los datos con número par.

Entonces, dados $a_0 , a_1 , \dots , a_{n-1}$ $n= 2^k$

Se obtiene como salida $A_j = \sum a_k e^{2\pi ijk/n}$ desde k=0 hasta k=n-1, para j = 0, 1, 2, .., n-1 a través de un algoritmo recursivo que responde al siguiente esquema:

1) Si n == 2
   $A_0 = a_0 + a_1$
   $A_1 = a_0 - a_1$

2) Obtener recursivamente los coeficientes de la transformada $a_0 , a_2 , a_{n-2} , \dots (a_1 , \dots a_{n-1})$.

   Los coeficientes se denotan por $B_0 , B_1 , \dots . B_{n/2}$ ($C_0 , C_1 , C_{n/2}$)

3) Para j=0 hasta j= n/2 -1
   $A_j = B_j + w_n^j C_j$
   $A_{j+n/2} = B_j + w_n^{j+n/2} C_j$

El método directo habitual tiene un coste temporal de $O(n^2)$. Aplicando D y V según el esquema indicado, puede disminuir a O(n log n)

Ejemplo 7: armado del fixture (calendario de juegos) para un campeonato

Se tienen n equipos participantes; cada equipo debe jugar una y sólo una vez con cada adversario, y , cada equipo participante debe jugar exactamente un partido diario. El problema consiste en confeccionar el calendario de partidos.
Si el número de equipos es una potencia de 2, es decir $n=2^k$, la solución se puede expresar a través de una matriz de n x n-1 en la cual, cada fila corresponda a un equipo, cada columna a un día y cada celda al número del equipo adversario con el cual ese equipo competidor debe enfrentarse el dia en cuestión.
Si se aplica la estrategia DyV los subproblemas se pueden establecer de la siguiente manera: el valor del umbral (el L) puede ser 2, dado que, cuando los equipos competidores sean 2, ya está resuelta la situación.

Si k>1,

- Se construyen, de forma independiente, dos subcalendarios de 2*k*-1 equipos participantes: el primero para los equipos numerados desde 1 hasta 2*k*-1, y el segundo para los participantes desde 2*k*-1+1 hasta 2*k*.

- Se determinan las competiciones cruzadas entre los equipos participantes de numeración inferior y los de numeración superior. Para esto se completa primero la parte de los equipos de numeración inferior de esta manera:
  - 1er equipo: compite en días sucesivos con los equipos participantes de numeración superior en orden creciente.
  - 2º equipo participante: toma la misma secuencia y realiza una permutación cíclica de un equipo participante
  - Se repite el proceso para todos los equipos de numeración inferior
  - Para los de numeración superior se hace lo análogo con los de inferior

Es decir,

```
for (j= diainferior ; j<= diasuperior; j=j+1)
  {   tabla[equipoinferior, j]  =  equipoinicial + j – diainferior;   }
for (i = equipoinferior + 1; i<= equiposuperior ; i= i+1)
  {   tabla [i, diainferior] = tabla [i – 1, diasuperior] ;}
for ( j= diainferior + 1; j<= diasuperior ; j= j + 1)
        { tabla[ i, j] = tabla [i – 1, j - 1] ;}
```

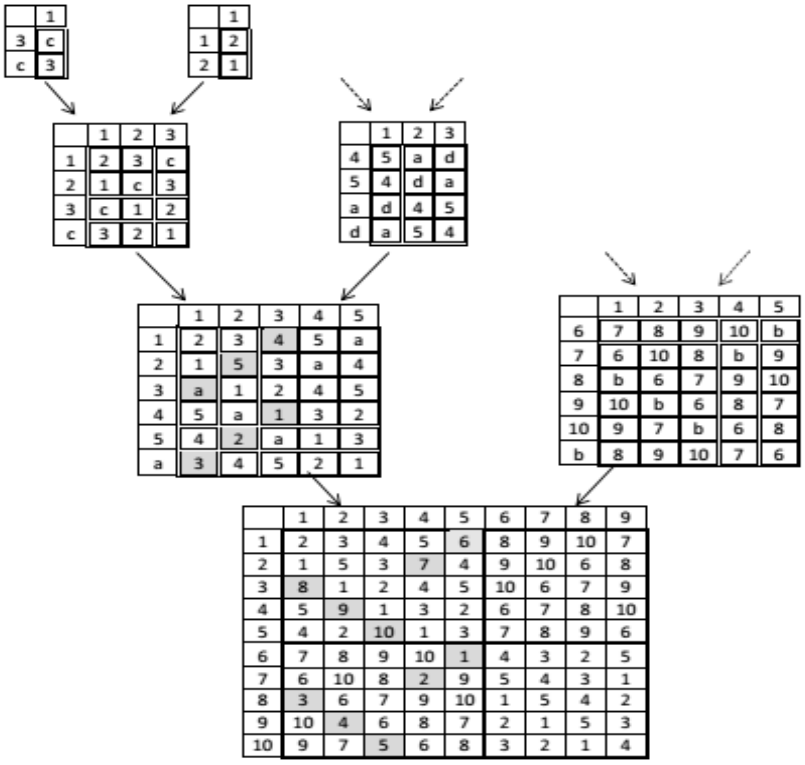
Si n no es potencia de 2, y es impar, se necesitan n días de competición, y no solamente n-1. Esto se puede resolver agregando un equipo participante ficticio numero n + 1. Este valor, n+1 es par, por lo cual se puede calcular el calendario de competiciones , y luego, se considerará que, para cada equipo participante **x** que deba competir el dia **y** con el equipo participante n + 1, el equipo **x** descansará el día **y**.

Si n es par se construye la tabla tal como se hizo par a $n = 2^k$.

Si n es impar se hace lo siguiente:
Se desarrolla la tabla para la primera parte de la competencia, para los n / 2 equipos participantes de numeración inferior, añadiendo un equipo ficticio.

- Se calcula la primera parte de la competencia para los otros n / 2 equipos participantes, agregando otro equipo ficticio.
- Para el dia j-esimo, con 1<=j<= n /2 se hace jugar entre si a los dos equipos participantes, uno de numeración inferior y otro de numeración superior, a los que les había tocado el j como día de descanso en las etapas anteriores.
- Se completan los restantes lugares de la tabla con competiciones cruzadas, análogamente a como se completó la tabla anteriormente.

Se muestra un ejemplo de desarrollo de tabla de competiciones:



Análisis de complejidad temporal:

La recurrencia indica
$T(n) = 2\,T(n/2) + n^2 / 4$
$T(1) = 1$
De lo cual se deduce $T(n)$ pertenece a $O(n^2)$