Técnica de diseño #1:
Divide y vencerás
(Divide-and-Conquer)

Análisis y Diseño de Algoritmos

Ing. Román Martínez M.

Caso 1

a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a
a a a a a b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b
b b b b b b b b b b b b b b b b b b

¿Cuántas *a's* hay antes de las *b's*?

Caso 2

1	31	61	91	121	151	181	211	241	271	301	331
2	32	62	92	122	152	182	212	242	272	302	332
3	33	63	93	123	153	183	213	243	273	303	333
5	35	65	95	125	155	185	215	245	275	305	335
8	38	68	98	128	158	188	218	248	278	308	338
9	39	69	99	129	159	189	219	249	279	309	339
10	40	70	100	130	160	190	220	250	280	310	340
12	42	72	102	132	162	192	222	252	282	312	342
15	45	75	105	135	165	195	225	255	285	315	345
23	53	83	113	143	173	203	233	263	293	323	353
24	54	84	114	144	174	204	234	264	294	324	354
27	57	87	117	147	177	207	237	267	297	327	357
28	58	88	118	148	178	208	238	268	298	328	358
29	59	89	119	149	179	209	239	269	299	329	359

¿Existe el dato 227?

Caso 3

- Obtener por medio de un algoritmo de orden logarítmico el valor de $\mathbf{a^n}$.

Divide y vencerás

- Las máximas latinas **divide et impera** (pronunciado: *dívide et ímpera*, «divide y domina»), **divide et vinces**, **divide ut imperes** y **divide ut regnes**, fueron utilizados por el gobernante romano Julio César y el emperador corso Napoleón.

Divide y vencerás

- Técnica para enfrentar la solución de problemas.
- Consiste en dividir un problema en 2 o más instancias del problema más pequeñas, resolver (conquistar) esas instancias, y obtener la solución general del problema, combinando las soluciones de los problemas más pequeños.
- Utiliza un enfoque de solución de tipo ***top-down***.
- Corresponde a una solución natural de tipo **recursivo**.

Casos que usan “Divide y vencerás”

- **Búsqueda**
- **Ordenamiento**
- **Multiplicación de matrices**
- **Aritmética de enteros grandes**

Ejemplo: Búsqueda secuencial

```
pos = 1;
while(pos<=n) and (arreglo[pos] < dato) do
   pos = pos + 1;
if (pos > n) or (arreglo[pos]<>dato) then
  pos = 0;
```

- Mejor caso: **1**
- Peor caso: ***n***
- Caso promedio: depende de probabilidades
  3n/4 + 1/4

Por lo tanto:

O(n)

Ejemplo: Búsqueda binaria

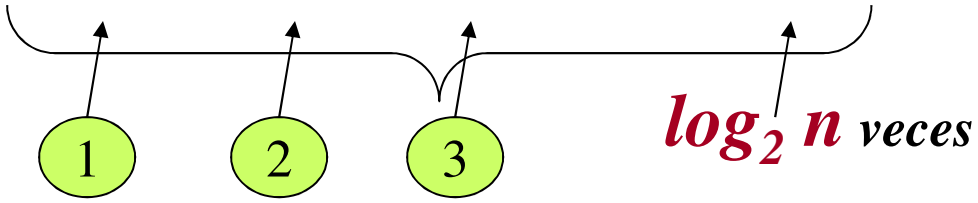
```
inicio =1; fin = n; pos = 0;
while (inicio<=fin) and (pos == 0) do
  mitad = (inicio+fin) div 2;
  if (x == arreglo[mitad]) then pos = mitad;
  else if (x < arreglo[mitad]) then fin = mitad-1
         else inicio = mitad+1;
```

- Operación Básica: **`x == arreglo[mitad]`**
- Mejor caso: **1**

Ejemplo: Búsqueda binaria

- Peor caso: No encontrar el dato
- Suponiendo que n es potencia de 2:

$$n/2 + n/4 + n/8 + \ldots + n/n$$



- Caso Promedio: Un análisis detallado lleva a encontrar la complejidad de: $\lfloor log_2\, n \rfloor \pm 1/2$
- Por lo tanto, el orden del algoritmo es: **O(log n)**

Búsqueda binaria

- Enfoque con la técnica de "divide y vencerás"
  SOLUCIÓN RECURSIVA

```
function busca (inicio, fin: index) : index
if (inicio > fin) then return 0;
else
   mitad = (inicio + fin) div 2;
   if (x == arreglo[mitad]) then return mitad;
   else if (x < arreglo[mitad]) then
           return(busca(inicio, mitad-1));
        else return(busca(mitad+1,fin));
```

Búsqueda binaria

- ¿Es diferente el comportamiento del algoritmo recursivo vs. el iterativo?
- **NO** en el contexto general del tiempo de ejecución…
- La complejidad de tiempo es diferente, pero por un valor constante…
- Por lo tanto, el orden es el mismo: **O(log n)**
- **SI** en el contexto de la complejidad de espacio del algoritmo, por el uso del stack en la recursividad.

Ordenamiento

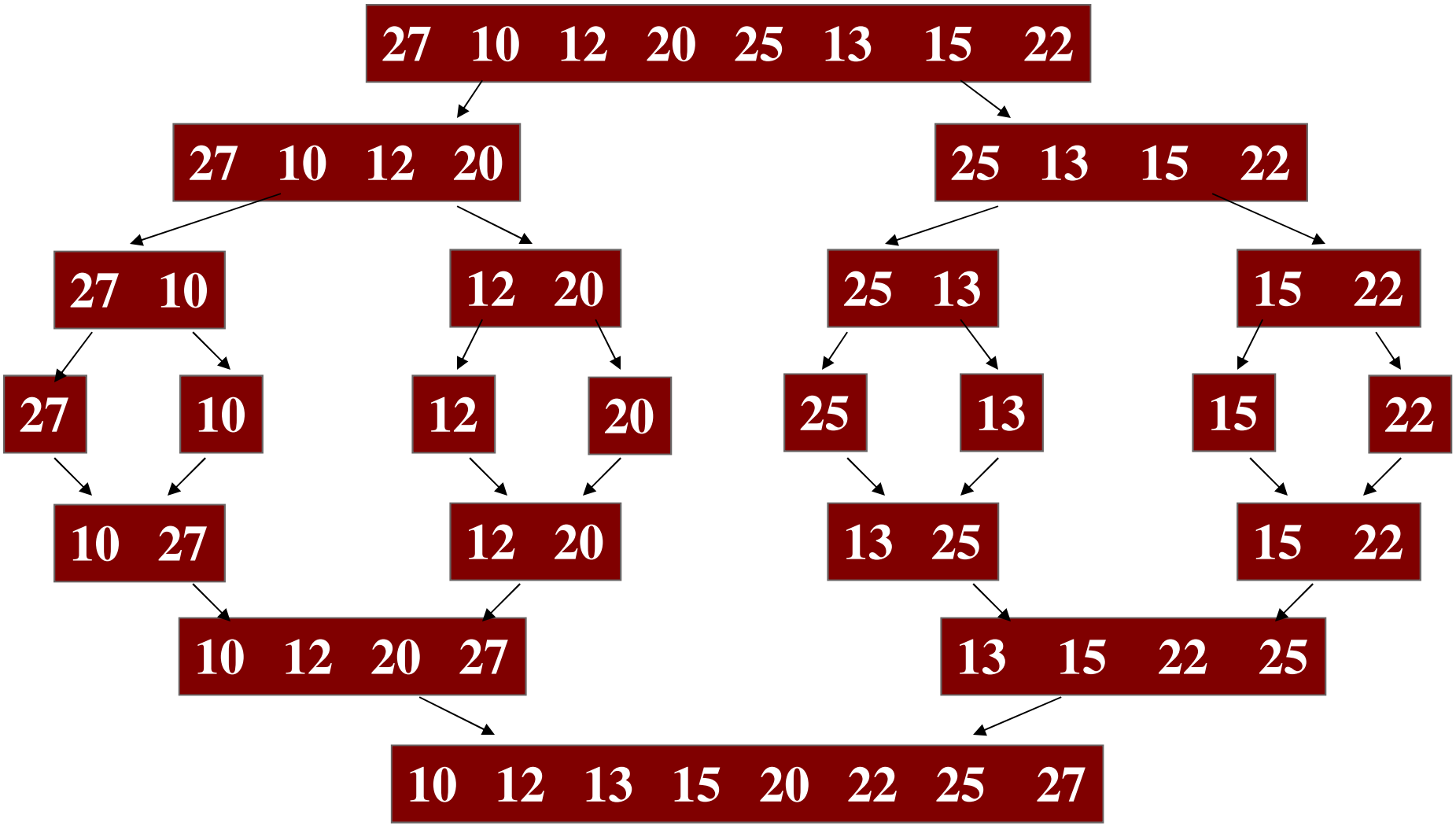
Merge Sort

Quick Sort

Merge Sort

- Divide el arreglo en 2 subarreglos.
- Se ordenan ambos subarreglos.
- Se forma el arreglo ordenado, considerando que se tienen 2 subarreglos ya ordenados.

Ejemplo: Merge Sort



Algoritmo: Merge Sort

```
Módulo MergeSort (inicio, fin)
if (inicio < fin) then
    mitad = (inicio+fin) div 2;
    MergeSort(inicio, mitad);
    MergeSort(mitad+1, fin);
    Une(inicio, mitad, fin);
```

Algoritmo: Une (Merge)

```
Módulo Une (inicio, mitad, fin)
i = inicio; j = mitad+1; k = inicio;
while (i<=mitad) and (j<=fin) do
    if (arreglo[i] < arreglo[j]) then
      aux[k] = arreglo[i]; i = i+1;
   else
      aux[k] = arreglo[j]; j = j+1;
   k = k +1;
if (i>mitad) then
  Mover elementos j a fin del arreglo al arreglo aux
  de k a fin;
else
  Mover elementos i a mitad del arreglo al arreglo
  aux de k a fin;
Copiar aux a arreglo;
```

Análisis del Merge Sort

- ¿Porqué no es un análisis “every-case”?
  - El algoritmo ***Une*** tiene comportamiento distinto dependiendo del caso.
- ¿Cuál es el peor caso si la operación de comparación es la que determina la complejidad del algoritmo?
  - Si ***n1*** y ***n2*** son los tamaños de los subarreglos…
  - Cuando ***n1-1*** datos del primer subarreglo son menores a los datos del otro subarreglo…
  - Y se hacen ***n1 - 1 + n2*** comparaciones, que equivalen a ***n - 1***

Análisis del Merge Sort

- Sea T(n) el peor tiempo para hacer el Merge Sort a un arreglo de n elementos…

$$\mathbf{T(n) = T(n/2) + T(n/2) + n\text{-}1}$$

Tiempo para ordenar el primer subarreglo	Tiempo para ordenar el segundo subarreglo	Tiempo para ejecutar el módulo UNE

$$\mathbf{T(n) = 2T(n/2) + n\text{-}1}$$

- La recurrencia se resuelve con: $n \log n - (n - 1)$
- Por lo tanto, el orden del peor caso es: **O(n log n)**

Resolviendo la recurrencia...

$T(n) = 2*T(n/2)$	$+ n-1$
$T(n) = 2*(2*T(n/4) + n/2-1)$	$+ n-1$
$T(n) = 4*T(n/4)$	$+ n-2 + n-1$
$T(n) = 4*T(n/4)$	$+ 2*n-3$
$T(n) = 4*(2*T(n/8)+(n/4-1))$	$+ 2*n-3$
$T(n) = 8*T(n/8)$	$+ n-4 + 2*n-3$
$T(n) = 8*T(n/8)$	$+ 3*n-7$
...	
$T(n) = 2^i*T(n/2^i)$	$+ i*n-(2^i-1)$

... hasta que $2^i = n$ y por lo tanto, $i = \log_2 n$

$T(n) = n*T(n/n)$	$+ \log_2 n*n-(n-1)$
$T(n) = n*T(1)$	$+ \log_2 n*n-(n-1)$
$T(n) = 0$	$+ \log_2 n*n-(n-1)$
$T(n) = \log_2 n*n-(n-1)$	

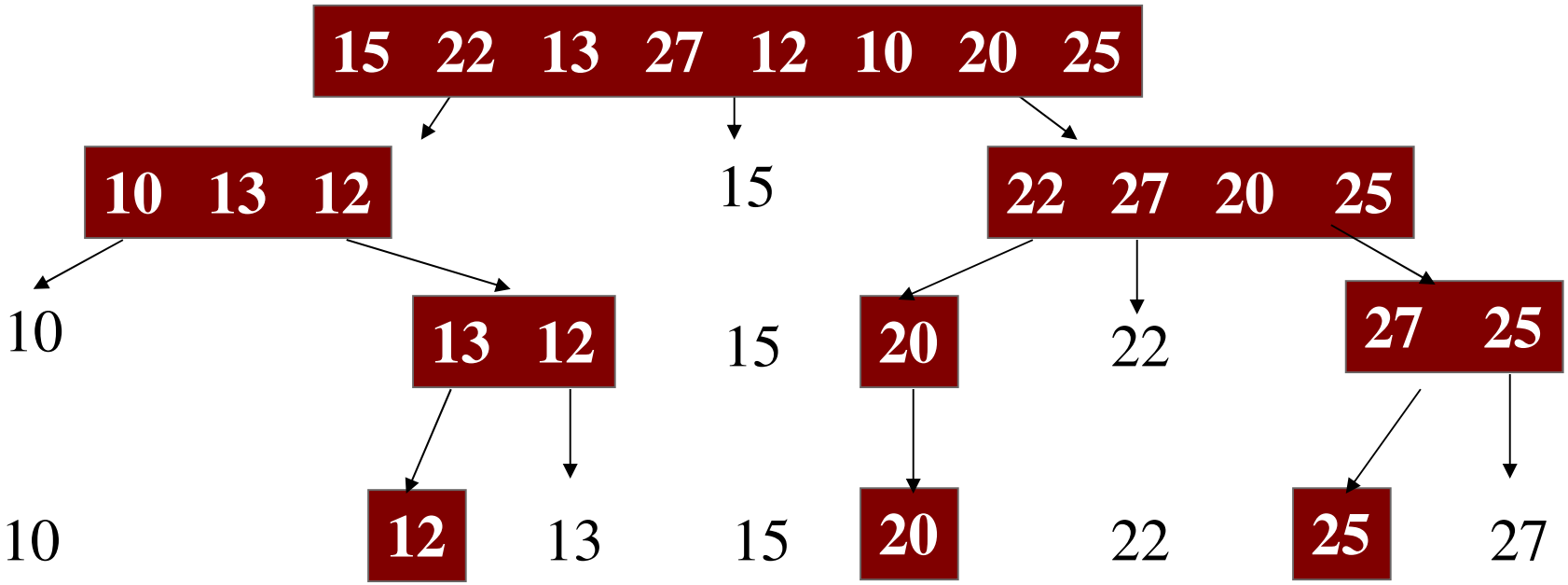
O(n log n)

Quick Sort

- Divide el arreglo en 2 particiones, una que contiene a los elementos menores a un elemento pivote, y otra que contiene a los elementos mayores al pivote.
- Se ordenan ambas particiones, y automáticamente se tiene todo el arreglo ordenado.
- La elección del elemento pivote es libre (por facilidad, se toma el primer elemento del arreglo).

Ejemplo: Quick Sort

Considerando que el primer elemento del arreglo es el pivote:



Algoritmo: Quick Sort

```
Módulo QuickSort (inicio, fin)
if (inicio < fin) then
   Partición(inicio, fin, pivote)
   QuickSort(inicio, pivote-1);
   QuickSort(pivote+1, fin);
```

Algoritmo: Partición

```
Módulo Partición (inicio, fin, pivote)
elempivote = arreglo[inicio]; j = inicio;
for (i = inicio+1; i<=fin; i++)
    if (arreglo[i] < elempivote) then
        j = j+1;
        Intercambia arreglo[i] con arreglo[j]
pivote = j;
Intercambia arreglo[inicio] con arreglo[pivote]
```

EJEMPLO: 15 22 13 27 12 10 20 25

Análisis del Quick Sort

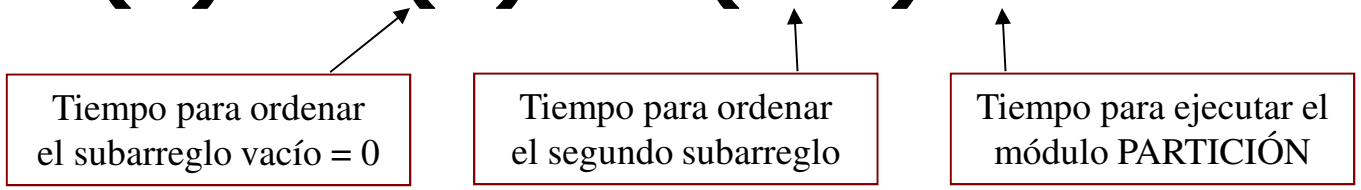
- ¿Porqué no es un análisis “every-case”?
  - La partición del arreglo es variable (depende del pivote).

- ¿Cuál es el peor caso si la operación de comparación (al hacer la Partición) es la que determina la complejidad del algoritmo?
  - Cuando la partición genera un subarreglo vacío y el otro con ***n-1*** datos
  - Se hacen ***n - 1*** comparaciones

Análisis del Quick Sort

- Sea T(n) el peor tiempo para hacer el Quick Sort a un arreglo de n elementos…

$$T(n) = T(0) + T(n-1) + n-1$$



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

- La recurrencia se resuelve con: $n * (n - 1) / 2$
- Por lo tanto, el orden del peor caso es: $O(n^2)$

Resolviendo la recurrencia...

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

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

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

…

$T(n) = T(n-i) + n-i + \ldots + n-2 + n-1$

…

$T(n) = T(n-n) + n-n + n-(n-1)\ldots + n-2 + n-1$

$T(n) = T(0) + 0 + 1 + \ldots + n-2 + n-1$

$T(n) = \textit{sumatoria de 1 a } n-1$

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

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

$O(n^2)$

Complejidad de los algoritmos

- **MERGE SORT**:
  - Peor caso: **O(n log n)**
  - Caso promedio: **O(n log n)**
- **QUICK SORT**:
  - Peor caso: **$O(n^2)$**
  - Caso promedio: **O(n log n)**

Algoritmo de Strassen para Multiplicar Matrices

- El análisis matemático de Strassen, descubrió que para:

$$\begin{bmatrix} c11 & c12 \\ c21 & c22 \end{bmatrix} = \begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix} \times \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix}$$

Existen los valores:

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

Tales que:

$\mathbf{c11 = m1 + m4 - m5 + m7}$
$\mathbf{c12 = m3 + m5}$
$\mathbf{c21 = m2 + m4}$
$\mathbf{c22 = m1 + m3 - m2 + m6}$

Algoritmo de Strassen

- Dividir cada una de las matrices en 4 submatrices, y resolver por el método de Strassen el problema.

$$\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \times \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix}$$

Cada submatriz es de $n/2$ X $n/2$

Análisis del algoritmo de Strassen

- La solución del caso de 2 X 2 requiere de **7** multiplicaciones y 18 sumas/restas...
- La solución tradicional, requiere de 8 multiplicaciones y 4 sumas…
- Aparentemente, no es significativo el beneficio…
- Pero ahorrar una multiplicación de matrices en el algoritmo de Strassen, a costa de más sumas o restas de matrices, tiene repercusiones significativas...

Análisis del algoritmo de Strassen

- Se considerará como la operación básica a medir a la MULTIPLICACIÓN de 2 datos.
- Sea T(n) el tiempo de una multiplicación de 2 matrices de n X n...
- Si el algoritmo de Strassen, requiere de 7 multiplicaciones, aplicadas a las matrices más pequeñas de $n/2$ X $n/2$, entonces...
- **T(n) = 7*T(n/2)** ...
- y la recurrencia se resuelve $T(n) = n^{\log 7} = n^{2.81}$

$$O(n^{2.81})$$

Aritmética de enteros grandes

- Un entero grande puede ser almacenado en una estructura en la que se guarde cada dígito del número.
- ¿Qué algoritmos están involucrados en la implementación de este tipo de dato?
  - Sumar 2 enteros grandes
  - Restar 2 enteros grandes
  - Multiplicar 2 enteros grandes
  - Dividir 2 enteros grandes, etc.

Orden de los algoritmos

- La suma y la resta, realizadas de manera tradicional, tienen un comportamiento lineal **O(n)**, donde n es la cantidad de dígitos del entero grande.
- La multiplicación en forma tradicional, tiene un comportamiento cuadrático **$O(n^2)$ en el peor caso**…
- Sin embargo, la multiplicación/división/residuo por una potencia de 10, tienen un comportamiento lineal **O(n)**.

Propuesta de mejora al algoritmo de la multiplicación

- Dividir el número en 2 números de tal manera que

$$e = x * 10^m + y$$

- EJEMPLO: 8,234,127 = $8234*10^3 + 127$
- Si los 2 números que se desean multiplicar se expresan de esta manera, se tiene que:
- $$e_1 * e_2 = (x_1*10^m + y_1) * (x_2*10^m + y_2)$$
$$= x_1x_2*10^{2m} + (x_1y_2 + x_2y_1)*10^m + y_1y_2$$
- De esta manera se hacen 4 multiplicaciones con enteros más pequeños...

Propuesta de mejora al algoritmo de la multiplicación

- El análisis de la propuesta, indica que el algoritmo sigue teniendo un orden cuadrático…
- Sin embargo, se puede eliminar una multiplicación con el siguiente análisis:
- $r = (x_1 + y_1) * (x_2 + y_2) = x_1x_2 + (x_1y_2 + x_2y_1) + y_1y_2$
- y por lo tanto: $(x_1y_2 + x_2y_1) = r - x_1x_2 - y_1y_2$
- y sustituyendo en: $x_1x_2*10^{2m} + (x_1y_2 + x_2y_1)*10^m + y_1y_2$
- $x_1x_2*10^{2m} + (r - x_1x_2 - y_1y_2)*10^m + y_1y_2$
- $x_1x_2*10^{2m} + ((x_1+y_1)*(x_2+y_2) - x_1x_2 - y_1y_2)*10^m + y_1y_2$

Algoritmo de la multiplicación

```
Function Multiplica (n1, n2: entero_grande):
   entero_grande;
n = cantidad de dígitos mayor entre n1 y n2.
if (n1 = 0 ) or (n2 = 0) return 0;
else if (n <= umbral ) return n1*n2 tradicional;
       else
             m = n div 2;
       x1 = n1 div 10^m; y1 = n1 mod 10^m;
       x2 = n2 div 10^m; y2 = n2 mod 10^m;
       r = Multiplica(x1+y1, x2+y2);
       p = Multiplica(x1, x2);   q = Multiplica(y1, y2);
       return (p X 10^2m + (r-p-q) X 10^m + q);
```

Límite en que resulta mejor realizar la operación en forma tradicional

$O(n^{1.58})$

Generalización de Divide y vencerás

Función DV(x)

if *x es suficientemente pequeño o sencillo*

    return *(solución tradicional al problema)*

else

    Descomponer x en casos más pequeños $x_1, x_2, \ldots x_m$

    for $i = 1$ ***to*** m ***do*** $y_i = DV(x_i)$

    Recombinar las y_i *para obtener la solución y de x*

    return *y*

Comportamiento general de algoritmos con DyV

$$\mathbf{O(n^k)} \text{ si } m < b^k$$

$$\mathbf{O(n^k \log n)} \text{ si } m = b^k$$

$$\mathbf{O(n^{\log_b m})} \text{ si } m > b^k$$

- donde m es la cantidad de subcasos más pequeños que se requieren para la solución (llamadas recursivas).
- b es el factor con que se divide el problema en casos más pequeños.
- k es un valor cualquiera para el análisis.

Condiciones para utilizar Divide y Vencerás

- Debe ser posible descomponer el problema en subproblemas.
- Debe ser posible recomponer las soluciones de una manera eficiente.
- Los subproblemas deben de ser, en lo posible, del mismo tamaño.
- **¿Cuándo NO utilizar DyV?**
  - Si el tamaño de los subproblemas es casi el mismo tamaño original.
  - Si la cantidad de subproblemas es casi la misma que el tamaño del problema.