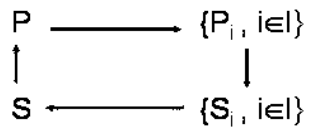
# 4

# Algoritmos Divide y Vencerás

## Introducción

Divide y Vencerás es una técnica para diseñar algoritmos que consiste en descomponer el caso del problema que queremos resolver en un número de subproblemas menores del mismo problema, resolviendo sucesiva e independientemente cada uno de ellos mediante un algoritmo que suponemos existe, al que en lo sucesivo referiremos como "básico", y después combinando las soluciones así obtenidas de modo que se obtenga la solución del problema inicial. Si el problema de partida lo notamos P, $\{P_i , i \in I\}$ representa la colección de subproblemas en la que dividimos P, $\{S_i , i \in I\}$ la de sus correspondientes soluciones y S la combinación de esas soluciones de los subproblemas, el siguiente diagrama refleja cómo se desarrollaría un algoritmo Divide y Vencerás




A pesar de la sencillez del método que describe este esquema, el desarrollo del mismo no está exento de dificultades, ya que hay diferentes escollos que deberemos tener previsto superar. Así, por ejemplo, es natural esperar que todos los subproblemas sean de la misma naturaleza entre sí, y que además esta coincida con la de P. Pero nada hay seguro acerca del número de subproblemas en el que deberemos dividir P. Solo hay datos basados en la experiencia que indican que la técnica Divide y Vencerás funciona mejor con un número pequeño de subproblemas que con uno grande. Tampoco sabemos nada sobre el tamaño que deberán tener los subproblemas. La práctica del Divide y Vencerás recomienda que cuanto más parecidos sean los tamaños de los subproblemas, mejor funciona el algoritmo. Pero no hay ningún resultado teórico que demuestre que este hecho siempre conduce a buenos resultados. En este punto es necesario destacar que cuando hablamos de dividir el problema en subproblemas, no estamos diciendo que esa división tenga que ser exacta y exhaustiva (lo que llamaríamos una partición) sino, más bien, que consideramos subproblemas de P, sin que ello suponga que la unión de todos ellos deba reproducir exactamente P. Podría pasar que la unión de todos los subproblemas reprodujera P, pero lo más normal será que de esa unión obtengamos algo más que P.

Entonces, supuesto que sabemos cuántos subproblemas tendremos y que tamaño tendrán estos, deberemos saber resolverlos, es decir, hallar las soluciones Si, $i \in I$, y lo que no es menos fácil, que podamos combinarlas entre sí para obtener una solución S que tenga sentido. Junto con todo eso, hay que confiar en que S se corresponda con la solución verdadera de P, y finalmente que todo este diagrama secuencial de pasos proporcione un algoritmo más eficiente que el básico que suponemos tenemos para resolver los subproblemas, y por tanto valido para resolver P si hemos dicho que aquellos deben ser de la misma naturaleza que este.
Bien, pues a pesar de todas estas dificultades y ambigüedades, cuando esta técnica puede aplicarse, proporciona algoritmos altamente eficaces. Comprobaremos esto con la siguiente ilustración. Supongamos que queremos resolver determinado problema P, y que para ello disponemos de un cierto algoritmo B (básico) de orden cuadrático, es decir, que una cierta implementación de B nos proporciona un tiempo

$$t_B(n) = bn^2$$

para resolver un caso de tamaño n.
Supongamos ahora que descubrimos que sería posible resolver tal caso descomponiéndolo en tres subcasos de tamaños n/2, resolviéndolos y combinando los resultados. Supongamos que la descomposición y la combinación de las soluciones parciales podemos llevarla a cabo mediante un algoritmo C que es lineal, es decir, de tiempo,

$$t_c(n) = kn$$

para una cierta implementación y determinada constante k. Usando a la vez el algoritmo básico inicial B y esta nueva idea, obtenemos un nuevo algoritmo N cuya implementación consume un tiempo.

$$t_N(n) = 3t_B(n/2) + t_c(n) = 3b(n/2)^2 + kn = (3/4)bn^2 + kn$$

El término $(3/4)bn^2$ domina cuando n es suficientemente grande, lo que significa que el algoritmo N es esencialmente un 25% más rápido que el algoritmo básico B. Pero, aunque esta es una mejora considerable, no hemos conseguido cambiar el orden del tiempo requerido, ya que el nuevo algoritmo N sigue siendo cuadrático.
Pero aún nos queda otra posibilidad para resolver más eficazmente P, y radica precisamente en la naturaleza de los subcasos, y por tanto de los subproblemas de P. En efecto, si los subcasos son pequeños, sería posible que el algoritmo B fuera el que mejor sirviera a nuestros intereses. Pero, cuando los subcasos son lo suficientemente grandes, ¿no sería mejor usar nuestro nuevo algoritmo N recursivamente?
Si hacemos esto obtenemos un nuevo algoritmo DV cuya implementación corre un tiempo,

$$t_{DV}(n) = \begin{cases} t_A(n) & \text{si } n \le n_0 \\ 3t_{DV}(n/2) + t_c(n) & \text{en otro caso} \end{cases}$$

donde $n_0$ es un valor umbral para el tamaño del caso con el que el algoritmo es llamado recursivamente.

.

Esta ecuación, que es similar a la de un ejemplo anterior, nos da un tiempo en el orden de $n^{lg3}$, que es muy aproximadamente $n^{1.59}$. Por tanto la mejora comparada con el orden de $n^2$ es sustancial, y lo grande que sea n es lo peor que este algoritmo puede tener. Veremos en la siguiente sección como elegir $n_0$ en la práctica. Aunque esta elección no afecta al orden del tiempo de ejecución del algoritmo, también nos interesa hacer la constante oculta que multiplica $n^{lg3}$ tan pequeña como sea posible.
Como la técnica Divide y Vencerás se aplica siempre de una forma sistemática, es susceptible de algoritmizarse. Así el método general divide y vencerás consistiría en lo siguiente

```
FUNCION DV(P)
    Si P es suficientemente pequeño o simple entonces devolver BASICO (P).
    Descomponer P en subcasos P(1), P(2), ..., P(k) más pequeños

            para i = 1 hasta k hacer S(i) = DV(P(i))
                recombinar las S(i) en S (solucion de P)
    Devolver (S)
```

donde BASICO, el subalgoritmo básico, se usa para resolver pequeños casos del problema en cuestión.
Como antes hemos comentado, el número de subcasos k, es usualmente bajo e independiente del caso particular a resolver. Cuando k=1, es difícil justificar el nombre de divide-y-vencerás, y en este caso a esta técnica se le llama de simplificación. Mencionemos también que algunos algoritmos de tipo divide y vencerás, no siguen exactamente el esquema precedente, sino que en su lugar, exigen que el primer subcaso se resuelvan antes de siquiera, se haya formulado el segundo subcaso.
En cualquier caso, aprovechando el carácter genérico de la anterior función algorítmica DV(P), podemos establecer que la eficiencia de un algoritmo Divide y Vencerás se calcula siempre a partir de una ecuación de la siguiente forma,

$$T(n) = \begin{cases} t(n) & \text{si } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{en otro caso} \end{cases}$$

donde a es el numero de subproblemas, n/b el tamaño de estos, D(n) el tiempo de dividir el problema en los sub-problemas y C(n) el tiempo de combinacion de las soluciones de los subproblemas. Si solo estuviéramos interesados en conocer el orden del algoritmo, nos bastaría con resolver esta ecuación sin necesidad de determinar las constantes que nos aparecerán en la solución. Sin embargo, si quisiéramos la solución exacta, que indudablemente estaría asociada a una implementación concreta, el cálculo de las constantes seria obligado.

## La determinación del umbral

Un algoritmo divide y vencerás debe evitar proceder recursivamente cuando el tamaño de los subcasos no lo justifique. En este caso es preferible usar el algoritmo

básico. Para ilustrar esto, consideremos de nuevo el algoritmo DV de la anterior sección, cuyo tiempo de ejecución está dado por,

$$t_{DV}(n) = \begin{cases} t_B(n) & \text{si } n \leq n_0 \\ 3t_{DV}(n/2) + t_C(n) & \text{en otro caso} \end{cases}$$

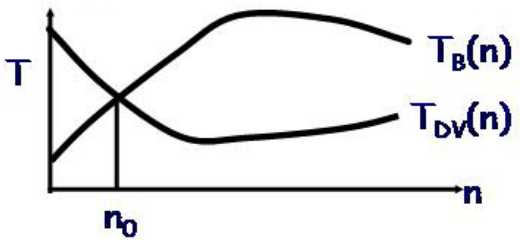
donde $t_B(n)$ es el tiempo requerido por el subalgoritmo básico y $t_C(n)$ es el tiempo consumido en hacer una descomposición y una recombinación.

Para determinar el valor del umbral $n_0$ que minimiza $t_{DV}(n)$, no es suficiente conocer que $t_B(n)$ sea cuadrático y que $t_C(n)$ sea lineal, ya que ese valor es evidente que ira asociado a la resolución de cada caso concreto, y a cada implementación particular.

En efecto, consideremos una implementación para la que los valores de $t_B(n)$ y de $t_C(n)$ estén dados, respectivamente, por $n^2$ y 16n milisegundos. Supongamos que tenemos para resolver un caso de tamaño 1024. Si el algoritmo procede recursivamente hasta obtener subcasos de tamaño 1, es decir, si $n_0 = 1$, consume más de media hora en resolver este caso. Esto es ridículo, ya que el subcaso puede resolverse en poco más de un cuarto de hora usando el subalgoritmo básico directamente, es decir tomando $n_0 = 1$. ¿Podríamos concluir que la técnica divide y vencerás nos permite pasar de un algoritmo cuadrático a un algoritmo cuyo tiempo de ejecución está en $O(n^{lg3})$, pero sólo a costa de un aumento en la constante oculta tan enorme que el nuevo algoritmo nunca es económico en casos que puedan resolverse en un tiempo razonable?. Pues la respuesta es no: en nuestro ejemplo, el caso de tamaño 1024 puede resolverse en menos de 8 minutos si elegimos apropiadamente el umbral $n_0$.

Este ejemplo muestra que la elección del umbral puede tener una influencia considerable en la eficiencia de un algoritmo de divide y vencerás. De todos modos, la elección del umbral es complicada ya que el mejor valor generalmente no depende sólo del algoritmo considerado, sino también de la implementación particular. ¿Cómo elegiremos entonces $n_0$?. Parece claro que una condición necesaria es que debemos tener $n_0 \geq 1$ para evitar la recursión infinita que resulta si la solución de un caso de tamaño 1 nos exige primero resolver otros cuantos casos del mismo tamaño.

Ahora bien, podemos plantear dos enfoques distintos para el cálculo del umbral: el teórico y el empírico. En efecto, dada una implementación particular, el umbral óptimo puede determinarse experimentalmente. Para ello, iríamos variando el valor del umbral y el tamaño de los casos usados para nuestros tests y estudiaríamos la implementación en un número de casos. Más concretamente, lo que podríamos obtener son dos curvas, para $t_B(n)$ y $t_{DV}(n)$, que deberían tener sentidos opuestos de crecimiento, como muestra la siguiente figura



Justificadamente hay que esperar que el algoritmo básico aumente su tiempo, hasta que se estabilice, con el tamaño del caso, mientras que la parte puramente recursiva del Divide y Vencerás deberá tener un comportamiento contrario, comportándose mal al principio para luego estabilizarse. Así, como el umbral es el tamaño del problema en el que ambos tiempos deben coincidir, el punto en que se corten ambas curvas nos dará el valor de $n_0$ a partir del cual esos sentidos se invierten y, por tanto el umbral que buscamos. Es obvio que este enfoque puede requerir una considerable cantidad de tiempo de computación. Sin embargo, un cálculo puramente teórico del umbral óptimo es difícilmente posible dado que varía de una implementación a otra.
Por tanto, el enfoque teórico-híbrido, que recomendamos consiste en determinar teóricamente la forma de las ecuaciones de recurrencia y entonces encontrar empíricamente los valores de las constantes usadas en estas ecuaciones para la implementación que manejamos. El umbral óptimo puede estimare entonces encontrando el valor de n con el que no hay diferencias, para un caso de tamaño n, si aplicamos el subalgoritmo básico directamente o si llevamos a cabo uno o más niveles de recursión.
Volviendo a nuestro ejemplo, el umbral óptimo puede encontrarse resolviendo la siguiente ecuación

$$t_B(n) = 3\, t_B(n/2) + t_C(n)$$

ya que en el valor umbral,

$$t_{DV}(n) = t_B(n/2)$$

Así, retomando los valores del ejemplo que considerábamos ($t_B(n) = n^2$, $t_C(n) = 16n$ milisegundos, y n = 1024), nos quedaría

$$n^2 = \tfrac{3}{4}\, n^2 + 16\, n \rightarrow n = \tfrac{3}{4}\, n + 16$$

y resolviendo tenemos $n_0 = 64$.
En cualquier caso es muy importante darse cuenta de que en general el valor del umbral óptimo que determinemos, ya de forma experimental o de manera teórica, será un valor aproximado puesto que el punto de corte mostrado en la anterior figura, o el valor obtenido en el último ejemplo no tienen por qué ser valores enteros. De aquí que en muchos casos, más que de valor umbral, se hable de intervalo umbral, para referirnos al hecho comentado

## Búsqueda Binaria

Los métodos de búsqueda, y en especial la búsqueda binaria (también conocida como dicotómica) son anteriores a los ordenadores. En esencia, la búsqueda binaria es el algoritmo que se usa para buscar una palabra en un diccionario o un nombre en una guía telefónica, y es probablemente la más simple aplicación del divide y vencerás. Desde tiempo inmemorial se conoce también este método por su eficiencia para encontrar en un determinado intervalo de valores reales las raíces de ecuaciones sencillas de la forma f(x) = 0.

Sea T[1...n] un array ordenado en orden creciente; es decir, $1 \leq i < j \leq n \Rightarrow T[i] \leq T[j]$, y sea x algún ítem. El problema consiste en encontrar x en el array T si realmente está en él. Si el ítem que estamos buscando no está en el array, entonces en su lugar queremos encontrar la posición donde podría estar insertado. Formalmente, queremos encontrar el índice i tal que $0 \leq i \leq n$ y $T[i] \leq x < T[i+1]$, con la convención lógica de $T[0] = -\infty$ y $T[n+1] = +\infty$. (Por convención lógica entendemos que estos valores no están, en efecto, presentes en el array como centinelas). La solución obvia es buscar linealmente, secuencialmente, cada elemento de T hasta llegar al final o hasta encontrar un ítem mayor que x.

**FUNCIÓN LINEAL** (T[1...n],x)
{búsqueda secuencial de x en un array T}
para i = 1 hasta n hacer
si T[i]> x entonces devolver I-1
devolver n

Este algoritmo claramente consume un tiempo $\Theta(1 + r)$, donde r es el índice retornado: este es $\Omega(n)$ en el peor caso y O(1) en el mejor caso. Si suponemos que todos los elementos de t son distintos, ese x está realmente en algún lugar del array, y que cada posición es equiprobable, entonces el número medio de pasadas al lazo es $(n^2 + 3n - 2)/2n$. En media, por tanto, así como en el peor caso, la búsqueda lineal consume un tiempo en $\Theta(n)$.
Para acelerar la búsqueda la técnica Divide y Vencerás sugiere buscar x, ya en la primera mitad del array, ya en la segunda. Para determinar cuál de estas búsquedas es la apropiada, comparamos x con un elemento en la mitad del array : si $x < T[1 + n/2]$, entonces la búsqueda de x se confina a T[1....n/2]; en otro caso, es suficiente buscar en T[1+n/2 ....n]. Obtenemos el siguiente algoritmo,

**FUNCIÓN BUSQUEDABINARIA** (T[1....n],x)
{búsqueda binaria de x en un array T}
si n=0 o x<T[1] entonces devolver 0
devolver buscabin (T, x)

**FUNCIÓN BUSCABIN** (T[i.. j],x)
{este procedimiento sólo se llama si $T[i] \leq x < T[j + 1]$ e $i \leq j$}
si i= j entonces devolver I
k = (i + j + 1) div 2
si x < T[k ] entonces devolver buscabin (T[i….k-1],x)
caso contrario devolver buscabin (T[k…j],x)

Está claro que el algoritmo consume un tiempo O(log n) para encontrar x en T[1…n] independientemente de la posición que ocupe x en T.
En efecto si T(n) es su tiempo de ejecución,

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

de donde fácilmente se calcula el tiempo logarítmico que hemos adelantado:

$$n = 2^k \Rightarrow t_k = t_{k-1} + c \Rightarrow (x-1) = 0 \Rightarrow t_k = A1^k \Rightarrow$$

$$T(n) = \log n$$

Pero como el algoritmo ejecuta sólo una de las dos llamadas recursivas, técnicamente es un ejemplo de simplificación más que de divide y vencerás.
Justamente porque la llamada recursiva está situada al final del algoritmo, es sencillo diseñar una versión iterativa.

```
FUNCIÓN  BINARITER (T[1...n, x)
    {búsqueda binaria iterativa de x en un array T}
    si n=0 o  x < T[1]  entonces devolver 0
    i = 1; j = n
    mientras  i < j hacer
       {T [i] ≤ x < T[j+1]}
       k = (i +j +1) div 2
       si  x < T[k] entonces  j = k-1
                        en caso contrario  i = k
    devolver i
```

Lo que pasa es que este algoritmo no es muy eficiente, porque la condición se debe comprobar demasiadas veces, aun cuando hayamos encontrado el elemento que buscamos, pero esa dificultad es fácil de evitar haciendo que inmediatamente después de encontrar el elemento que andamos buscando, se produzca una salida del lazo:, como efectúa el siguiente algoritmo:

```
FUNCIÓN  BINARITER+ (T[1..n],x) {variante de la búsqueda iterativa}
    si n = 0 o  x < T[1]  entonces devolver 0
    i = 1; j = n
    mientras  i< j hacer
       {T [i] ≤ x < T[j+1] }
       k = (i +j ) div 2
       caso que   x < T[k] :  j = k-1
                              x ≥ T [k + 1] : i = k+1
       otrocaso  i, j = k
    devolver i
```

En general, los algoritmos Divide y Vencerás para la ordenación se ajustan al esquema que se desarrolla a continuación, que representa el enfoque genérico a seguir para resolver cualquier problema de esta naturaleza[1]

**ALGORITMO GENERAL DE ORDENACION DIVIDE Y VENCERAS**

```
Iniciar Ordenar(L)
        Si L tiene longitud mayor de 1 Entonces
                Comienzo
                  Partir la lista en dos listas, izquierda y derecha
                        Iniciar Ordenar(izquierda)
                        Iniciar Ordenar(derecha)
                        Combinar izquierda y derecha
                Fin
```

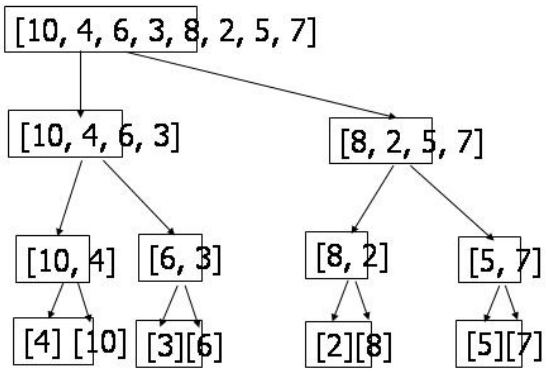
[1] Se trata de un esquema algoritmo de tipo general que habrá que especificar para resolver cada problema particular.

## Ordenación por Mezcla

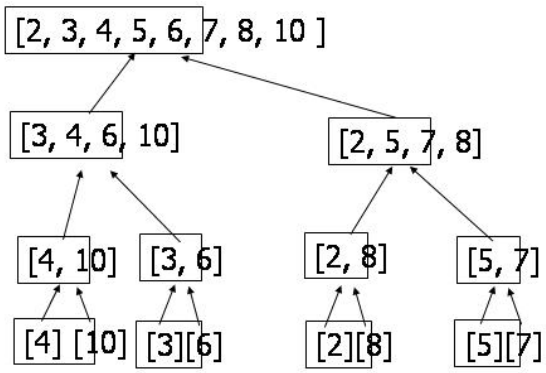
Sea T [1..n] un array de n elementos para el que existe un orden total. Estamos interesados en el problema de ordenar estos elementos en orden ascendente. Ya hemos visto que el problema puede resolverse ordenando por diversos métodos (selección, inserción, etc.). También vimos con anterioridad este método de ordenación por mezcla aunque sin especificar la técnica de diseño en la que se basaba. El enfoque obvio de Divide y Vencerás para este problema, de acuerdo con el algoritmo genérico que antes hemos escrito, consiste en separar el array T en dos partes cuyos tamaños sean tan próximos como sea posible, ordenar estas partes por llamadas recursivas, y entonces mezclar las soluciones de cada parte, siendo cuidadosos de preservar el orden.
La técnica Divide y Vencerás, por tanto, proporciona el siguiente algoritmo,

```
ALGORITMO MEZCLA (T[1..n])
    {ordena un array T en orden creciente}
    si n es pequeño entonces inserción (T)
    caso contrario U[1..n  div  2], V[1..(n+1)  div  2]
        U  = T[1..n  div  2]
        V = T[1 + (n  div  2)..n]
        mezcla (U); mezcla (V)
        combina (T, U, V)
```

donde inserción (T) es el algoritmo de ordenación por inserción y combina (T, U, V) mezcla en un solo array ordenado T, los arrays U y V que ya están ordenados.
Este algoritmo de ordenación es la ilustración perfecta de todas las facetas del Divide y Vencerás. Cuando el número de elementos a ordenar es pequeño, se usa un algoritmo relativamente simple. Por otro lado, cuando está justificado por el número de elementos, el Algoritmo Mezcla separa el caso en dos subcasos de tamaño mitad, resuelve cada uno de ellos recursivamente, y entonces los combina en dos medios arrays ordenados para obtener la solución del caso original.
Por ejemplo, como ilustración del método, supongamos que queremos ordenar el vector de números enteros (10, 4, 6, 3, 8, 2, 5, 7) empleando este algoritmo. La partición del vector en dos mitades se desarrollaría conforme al siguiente esquema



Y la operación de mezcla para k = 2 produciría




En cuanto a su eficiencia, sea t(n) el tiempo tomado por este algoritmo para ordenar un array de n elementos. Separar T en U y V consume un tiempo lineal y la mezcla final también consume un tiempo lineal, por tanto

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

y, como ya vimos anteriormente, el algoritmo tiene una eficiencia O(n log n).
Es importante comentar dos aspectos. En primer lugar, el hecho de que la suma de los tamaños de los subcasos sea igual al tamaño del caso original, no es típico de los algoritmos que se obtienen usando divide y vencerás. En segundo lugar, y como al principio de la descripción de esta técnica comentamos, el que el caso original esté dividido en subcasos cuyos tamaños son iguales, o tan parecidos como sea posible, es crucial de cara a obtener un algoritmo eficiente ya que si, por ejemplo, decidimos separar T en un array U con n-1 elementos y en otro V con solo un elemento, es decir, en dos subproblemas completamente descompensados, si t'(n) es el tiempo requerido por esta variante para ordenar n ítems, tendríamos

$$t'(n) = t'(n-1) + t'(1) + an$$

así que, como la ecuación característica seria $(x-1)^3 = 0$ (p(n) = n y b = 1), se deduce que t'(n) es $O(n^2)$, un tiempo mucho peor que el logarítmico anterior. Por tanto, se hace patente que el simple hecho de olvidar equilibrar los tamaños de los subcasos puede ser desastroso para la eficiencia de un algoritmo obtenido usando divide y vencerás.

## Algoritmo Quicksort

El algoritmo de ordenación inventado por Hoare, usualmente conocido por el nombre de "quicksort", también se basa en la idea del divide y vencerás. A diferencia de la ordenación por mezcla, la parte no recursiva del trabajo a realizar se gasta construyendo los subcasos en lugar de combinar sus soluciones. En una primera etapa este algoritmo elige como pivote uno de los ítems en el array a ordenar. Entonces el array se particiona a cada lado del pivote: los elementos son

colocados de tal forma que los que son mayores que el pivote se ponen a su derecha, mientras que todos los demás se ponen a su izquierda. Si ahora las dos secciones del array a cada lado del pivote se ordenan independientemente mediante llamadas sucesivas de los algoritmos, el resultado final es un array completamente ordenado que no necesita posteriores etapas de mezcla.

Para equilibrar los tamaños de los dos subcasos que hay que ordenar, podríamos usar el elemento mediano como pivote. Desgraciadamente, encontrar la mediana consume tanto tiempo que es peor. Así, como no hay definido acerca de la forma de elegir el pivote, salvo que la elección condiciona el tiempo de ejecución, cada uno podemos diseñar hoy mismo nuestro propio algoritmo Quicksort (otra cosa es que funcione mejor que los que ya hay...).

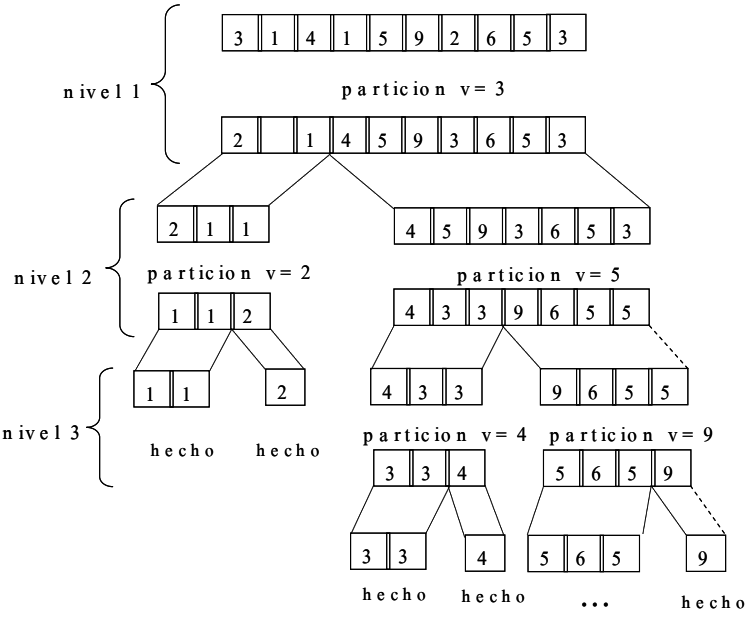
El pivote puede ser cualquier elemento en el dominio, pero no necesariamente tiene que estar en S. Así,

a) Podria ser la media de los elementos seleccionados en S
b) Podria elegirse aleatoriamente, pero la funcion la función de selección aleatoria consume tiempo, que habria que añadirselo al tiempo total del algoritmo
c) Otros pivotes pueden ser la mediana de un minimo de tres elementos (sin ninguna justificación teórica), el elemento medio de S o el elemento en la posicion central del array, dividiendo este en dos mitades

Entre todas las posibilidades, la que produce mejores resultados es la que inicialmente se proponía en el algoritmo, como se diseñó originalmente, es decir, escoger como pivote el mayor de los dos primeros elementos del array como pivote.

Antes de desarrollar el algoritmo, veamos el funcionamiento sobre el siguiente ejemplo. Consideremos el array (3, 1, 4, 1, 5, 9, 2, 6, 5, 3).

Entonces:




Con estos requisitos, el algoritmo es el siguiente,

**ALGORITMO QUICKSORT (T)**
Comienzo
Si TAMAÑO(T) ≤ q (umbral) Entonces INSERCION(T)
En caso contrario
    Elegir como pivote p el mayor de los dos primeros elementos del array
    Partir T en (T1,T2,T3) de modo que
        1. ∀ x∈T1, y∈T2, z∈T3 se verifique x<p<z e y=p
        2. TAMAÑO(T1) < TAMAÑO(T) y TAMAÑO(T3) < TAMAÑO(T)
        QUICKSORT(T1) // ordena recursivamente particion izquierda
        QUICKSORT(T3) // ordena recursivamente particion derecha
        Combinacion: T = S1 || S2 || S3 {S2 es el elemento intermedio entre cada mitad ordenada}
Fin

No es difícil diseñar un algoritmo en tiempo lineal para el pivoteo. Sin embargo, es crucial en la práctica que la constante oculta sea pequeña. Si admitimos que el procedimiento de pivoteo es lineal, es fácil imaginarse que si el Quicksort lo llamamos para T[1...n], y elegimos como peor caso que el pivote es el primer elemento del array, entonces el tiempo del anterior algoritmo sería

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

Que evidentemente se corresponde con una eficiencia de orden cuadrático.
Sea p = T[i] el pivote. Una buena forma de pivotear consiste en explorar el array T[i..j] sólo una vez, pero comenzando desde ambos extremos. Los punteros k y l se inicializan en i y j+1 respectivamente. El puntero k se incrementa, entonces, hasta que T[k] > p, y el puntero l se disminuye hasta que T[l] ≤ p. Ahora T[k] y T[l] están intercambiados. Este proceso continúa mientras que k < l. Finalmente, T[i] y T[l] se intercambian para poner el pivote en su posición correcta.

**ALGORITMO PIVOTE (T[i..j])**
    {permuta los elementos en el array T[i..j] de tal forma que al final i ≤ l ≤ j, los elementos de T[i..l-1] no son mayores que p, T[l] = p, y los elementos de T[l+1..j] son mayores que p, donde p es el valor inicial de T[i]}
Comienzo
    p = T[i]
    k = i; l = j+1;
    repetir k = k+1 hasta T[k] > p o k ≥ j
    repetir l = l-1 hasta T[l] ≤ p
    mientras k < l hacer
    intercambiar T[k] y T[l]
    repetir k = k+1 hasta T[k] > p
    repetir l = l-1 hasta 1T[l] ( p
    intercambiar T[i] y T[l]
Fin

El algoritmo Quicksort es ineficiente si ocurre sistemáticamente que en muchas llamadas recursivas los subcasos a ordenar T[i..l-1] y T[l+1..j] están muy

desequilibrados. Pero además, en el peor caso el quicksort requiere tiempo cuadrático.
Por otro lado, si el array a ordenar inicialmente está en orden aleatorio, podemos admitir que muchas veces los subcasos a ordenar estén suficientemente bien equilibrados. Para determinar el tiempo promedio que requiere el quicksort para ordenar un array de n elementos, suponemos que todos ellos son distintos y que cada una de las n! posibles permutaciones iniciales de los elementos tiene la misma probabilidad de darse. Sea t(m) el tiempo promedio que consume una llamada a quicksort (T[a+1..a+m]) para $0 \leq m \leq n$ y $0 \leq a \leq n-m$.
El pivote elegido por el algoritmo está situado con igual probabilidad en cualquier posición con respecto a los demás elementos de T. El valor de l que devuelve el algoritmo de pivoteo después de la llamada inicial pivote (T[1..n]) puede ser, por tanto, cualquier entero entre 1 y n, teniendo cada valor probabilidad 1/n. Esta operación de pivoteo consume un tiempo de orden exacto ((n). Quedan por ordenar recursivamente dos subarrays de tamaños l-1 y n-l, respectivamente. El tiempo promedio requerido para ejecutar estas llamadas recursivas es t(l-1) + t(n-l).
Por tanto,

$$T(n) = an + (1/n) \sum_{l-1..n}[t(l-1) + t(n-1)]$$

A partir de esta ecuación puede demostrarse que el quicksort puede ordenar un array de n elementos distintos en un tiempo promedio en O(n log n). Como además puede demostrarse que la constante oculta es en la práctica menor que la correspondiente del heapsort o de la ordenación por mezcla, si ocasionalmente puede tolerarse un tiempo de ejecución grande, éste es el algoritmo de ordenación preferible entre todos los presentados en estos temas.

## Aritmética con Enteros Grandes

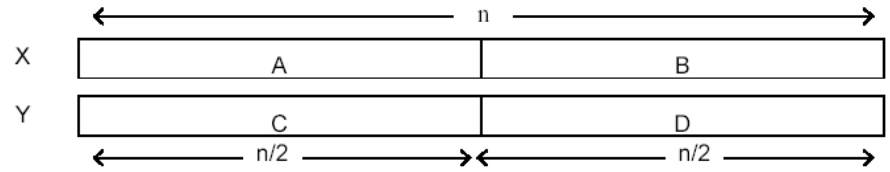
En muchos de los análisis anteriores hemos supuesto que la suma y la multiplicación eran operaciones elementales, es decir, que el tiempo requerido para ejecutar estas operaciones estaba acotado superiormente por una constante que sólo depende de la velocidad de los circuitos del ordenador que se esté usando. Esto sólo es razonable si el tamaño de los operandos es tal que pueden ser manejados directamente por el hardware.
Pero, para algunas aplicaciones tenemos que considerar enteros muy grandes. La representación de estos números en coma flotante no es útil salvo que estemos interesados sólo en el orden de magnitud y en algunas de las más importantes cifras de nuestros resultados. Si los resultados tienen que ser calculados exactamente y valen todas las cifras, estamos obligados a implementar las operaciones aritméticas en el software. Esto fue, por ejemplo, necesario cuando se calcularon los primeros 134 millones de dígitos de $\pi$ en 1987.
Sean X e Y dos enteros de n dígitos decimales para multiplicar. Recordemos que el algoritmo para la multiplicación de esos dos enteros de n bits (dígitos) que usualmente empleamos, supone calcular n productos parciales de tamaño n, y así es

un algoritmo $O(n^2)$, si contamos las multiplicaciones simples y las adicionales como una etapa:

```
                x = 1 0 1 1 0 1 0 1
                y = 1 0 1 0 0 1 1 0
-----------------------------------
                    1 0 1 1 0 1 0 1
                  1 0 1 1 0 1 0 1
              1 0 1 1 0 1 0 1
          1 0 1 1 0 1 0 1
-----------------------------------
      1 1 1 0 1 0 1 0 1 0 1 1 1 1 0
```

```
        x = 1 0 1 1
        y = 0 1 1 0
-------------------
            1 0 1 1
          1 0 1 1
-------------------
      1 0 0 0 0 1 0
```

Un enfoque Divide y Vencerás para este problema supondría romper X e Y en dos enteros de tamaños n/2 cada uno (suponiendo por simplicidad que n es par), de la siguiente forma




con lo que quedaría

$$X = A \cdot 2^{n/2} + B$$

$$Y = C \cdot 2^{n/2} + D$$

Por ejemplo, si X = 13, entonces

$$13 = \underset{3}{11} \quad \underset{1}{01} \quad \Rightarrow n = 4$$

es decir A = 3 y B = 1, con lo que

$$13 = A \cdot 2^{n/2} + B = 3 \cdot 2^{4/2} + 1 = (11)_{10} \cdot 2^2 + (01)_{10}$$

Así,

$$X \cdot Y = A \cdot C \cdot 2^n + (AD + BC) \cdot 2^{n/2} + BD \qquad (1)$$

Si evaluamos XY de este modo directo tenemos que hacer 4 multiplicaciones de enteros de n/2 bits (AC, AD, BC, BD), tres adiciones de enteros con, a lo sumo, 2n bits, y dos multiplicaciones por $2^n$ y $2^{n/2}$. Como estas adiciones y multiplicaciones por potencias de dos son etapas en O(n), podemos escribir la siguiente recurrencia para T(n), el número total de operaciones que son necesarias para multiplicar dos enteros de n bits de acuerdo con (1).

$$T(1) = 1$$

$$T(n) = 4T(n/2) + cn \qquad (2)$$

Para resolver la recurrencia, tomando c = 1 sin pérdida de generalidad, se deduce que T(n) es O(n2), o resolviéndola por expansión,

$T(n) = 4T(n/2) + cn = 4(4T(n/2^2) + cn/2) + cn = 4^2T(n/2^2) + 2cn + cn =$

$= 4^3T(n/2^3) + 2^2cn + 2cn + cn = \ldots = 4^mT(1) + (2^{m-1} + \ldots 1)\, cn =$

$= 2^m2^m + (2^m - 1)cn = O(n^2)$

En caso de que la fórmula (1) se use para multiplicar enteros, la eficiencia asintótica no es mayor que la del método elemental que generalmente empleamos. Pero recordemos que para ecuaciones como la (2), tendremos una mejora asintótica si disminuimos el número de subproblemas. Puede resultar increíble que podamos hacer eso, pero consideremos la siguiente fórmula para multiplicar X e Y,

$$X \cdot Y = AC \cdot 2^n + [(A-B)(D-C) + AC + BD)] \cdot 2^{n/2} + BD \qquad (3)$$

de donde es fácil deducir que T(n) es $O(n^{lg3})$, es decir, T(n) es $O(n^{1,59})$. Un algoritmo que lleva a cabo esta multiplicación puede ser el siguiente,

**FUNCION MULT** (X, Y)
s = signo (X) · signo (Y)
X = Abs(X); Y = Abs(Y)
si n= 1 entonces
si (X = 1) y (Y = 1) entonces devolver (s)
si no devolver (0)
si no begin
A = n/2 bits de la parte izquierda de X;
B = n/2 bits de la parte derecha de X
C = n/2 bits de la parte izquierda de Y;
D = n/2 bits de la parte derecha de Y
M1 = mult(A, C, n/2); M2 = mult(A-B, D-C, n/2); M3 = mult(B, D, n/2)
return ($s \cdot (M1 \cdot 2^{n/2} + (M1 + M2 + M3) \cdot 2^{n/2} + M3)$)

Como ejemplo, si

$x = 1\ 0\ 1\ 1 \qquad a = 1\ 0 \qquad b = 1\ 1$
$y = 0\ 1\ 1\ 0 \qquad c = 0\ 1 \qquad d = 1\ 0$

$u = (a+b)(c+d) = (1\ 0\ 1)(1\ 1) = 1\ 1\ 1\ 1$
$v = ac = (1\ 0)(0\ 1) = 1\ 0$
$w = bd = (1\ 1)(1\ 0) = 1\ 1\ 0$
$z = u - v - w = 1\ 1\ 1$
$xy = v2^4 + z2^2 + w = 1\ 0\ 0\ 0\ 0\ 0 + 1\ 1\ 1\ 0\ 0 + 1\ 1\ 0 = 1\ 0\ 0\ 0\ 0\ 1\ 0$

Hemos demostrado por tanto cómo es posible multiplicar dos enteros de n dígitos en un tiempo $O(n^{lg3})$, es decir, $O(n^{1,59})$. Pero las constantes ocultas son tales que este algoritmo sólo es interesante en la práctica cuando n es bastante grande. Una buena implementación probablemente no usaría la base 2, si no más bien la mayor base para la que el hardware permitiera que dos "dígitos" se multiplicaran directamente.
La multiplicación no es la única operación interesante con números enteros. La división entera, las operaciones de módulo y los cálculos de partes enteras de una raíz cuadrada pueden llevarse a cabo todas ellas en un tiempo cuyo orden es el mismo que el requerido para la multiplicación. Algunas otras operaciones

importantes, tales como el cálculo del máximo común divisor, de modo inherente pueden ser más duras de manejar: Aquí no las trataremos.

## Multiplicación de Matrices

Sean A y B dos matrices nxn que queremos multiplicar, y sea C su producto. El algoritmo clásico se hace directamente a partir de la definición,

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Cada elemento en C se calcula en un tiempo en $\Theta(n)$, supuesto que la adición y multiplicación de escalares son operaciones elementales. Como hay $n^2$ elementos que calcular para obtener C, el producto de A y B puede calcularse en un tiempo en $\Theta(n^3)$. El algoritmo, efectivamente, no sería más que un anidamiento de tres lazos de longitud n.
Desde el punto de vista del Divide y Vencerás la multiplicación puede hacerse como se muestra a continuación:

$$\underset{\Uparrow\ C}{\begin{pmatrix} r & s \\ t & u \end{pmatrix}} = \underset{\Uparrow\ A}{\begin{pmatrix} a & b \\ c & d \end{pmatrix}} \underset{\Uparrow\ B}{\begin{pmatrix} e & g \\ f & h \end{pmatrix}} = \begin{pmatrix} ae+bf & ag+bh \\ ce+df & cg+dh \end{pmatrix}$$

Como se ve, esta formulacion divide una matriz nxn en dos matrices de tamaños n/2 x n/2, con lo que se divide el problema en 8 subproblemas de tamaños n/2. Así la eficiencia del método se calcularía a partir de la siguiente recurrencia

$$T(n) = \begin{cases} b & \text{si } n = 1 \\ 8T(n/2) + bn^2 & \text{si } n > 1 \end{cases}$$

A partir de la cual, es evidente que T(n) sigue siendo $O(n^3)$ y que no ganamos nada con la aplicación de esta tecnica de diseño.
Hacia finales de los 60, Strassen mejoró este algoritmo. La idea básica es similar a la usada en el algoritmo de divide y vencerás para multiplicar enteros grandes. Vamos a mostrar primero que dos matrices 2x2 pueden multiplicarse usando menos de las ocho multiplicaciones escalares aparentemente requeridas por la definición. Sean,

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{ y } B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

dos matrices que queremos multiplicar. Consideremos las siguientes operaciones,

$$m_1 = (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11})$$

$$m_2 = a_{11} b_{11}$$

$$m_3 = a_{12} b_{21}$$

$$m_4 = (a_{11} - a_{21})(b_{22} - b_{12})$$

$$m_5 = (a_{21} + a_{22})(b_{12} - b_{11})$$

$$m_6 = (a_{12} - a_{21} + a_{11} - a_{22}) b_{22}$$

$$m_7 = a_{22}(b_{11} + b_{22} - b_{12} - b_{21})$$

Es fácil verificar que el producto pedido de A y B está dado por la siguiente matriz,

$$\begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Por tanto es posible multiplicar dos matrices 2x2 usando sólo siete multiplicaciones escalares. A primera vista, este algoritmo no parece muy interesante: usa un gran número de sumas y restas en comparación con el algoritmo clásico que usa cuatro adiciones, pero como su eficiencia se calcula a partir de

$$T(n) = \begin{cases} b & \text{si } n = 1 \\ 7T(n/2) + bn^2 & \text{si } n > 1 \end{cases}$$

es inmediato encontrar con el método de la función característica que la eficiencia es $O(n^{2.81})$. Así mismo, si resolviéramos esta ecuación por el método de expansión, se obtendría

$$\begin{aligned} T(n) &= 7T(n/2) + bn^2 \\ &= 7(7T(n/4) + b(n/2)^2) + bn^2 \\ &= 7^2 (7T(n/8) + b(n/4)^2) + (7/4 + 1)bn^2 \\ &= 7^m T(1) + ((7/4)^{m-1} + \ldots + (7/4) + 1)bn^2 \\ &= ((7/4)^m + \ldots + (7/4) + 1)bn^2 \\ &= ((7/4)^m - 1)bn^2 / ((7/4 - 1) \\ &= O(7^m) = O(n^{\log 7}) = O(n^{2.81}) \end{aligned}$$

con lo que volvemos a comprobar que el método de Strassen, efectivamente, rebaja el tiempo de ejecución para la multiplicación de matrices.

Si ahora reemplazamos cada elemento de A y B por una matriz nxn, obtenemos un algoritmo que puede multiplicar dos matrices 2nx2n llevando a cabo siete multiplicaciones de matrices nxn, así como un numero de sumas y restas de matrices nxn. Esto es posible porque el algoritmo básico no tiene en cuenta la conmutatividad de la multiplicación escalar. Dado que las adiciones de matrices pueden ejecutarse mucho más rápidamente que las multiplicaciones de matrices, las pocas más sumas

adicionales comparadas con el algoritmo clásico están más que compensadas ahorrando una multiplicación, dado que n es suficientemente grande.
A partir de aquí es fácil demostrar que es posible multiplicar dos matrices nxn en un tiempo en $O(n^{2.81})$, teniendo en cuenta que cuando el tamaño de las matrices no es potencia de 2, podemos orlarlas para conseguir el tamaño apropiado.
Un buen número de investigadores han intentado mejorar la constante ω tal que es posible multiplicar dos matrices nxn en un tiempo en $O(n^{\omega})$. Casi diez años antes que Strassen, Pan descubrió un algoritmo más eficiente basado en el divide y vencerás: encontró un modo de multiplicar dos matrices 70x70 con sólo 143.600 multiplicaciones escalares. (Nótese que 703 = 343.000 ). Numerosos algoritmos asintóticos más y más eficientes han sido descubiertos después. Uno de los algoritmos de multiplicación de matrices asintóticamente más rápido que se conoce hasta el momento puede multiplicar dos matrices nxn en un tiempo en O(n2.376); fue descubierto por Coppersmith y Winograd en septiembre de 1986. Debido a las constantes ocultas, sin embargo, ninguno de los algoritmos encontrados después del de Strassen es de mucho uso práctico.

## Multiplicación de Polinomios

Una consecuencia inmediata del algoritmo de multiplicación de enteros grandes, es su aplicación a la multiplicación de polinomios.
Supongamos dos polinomios:

$$P(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_{n-1} x^{n-1}$$
$$Q(x) = q_0 + q_1 x + q_2 x^2 + \ldots + q_{n-1} x^{n-1}$$

con el exponente n par. Evidentemente, hay n coeficientes y el grado de cada polinomio es n– 1. Para multiplicar estos dos polinomios, al amparo de cómo multiplicamos enteros grandes, la técnica Divide y Vencerás sugiere dividirlos por la mitad, quedando de la siguiente manera:

P(x):

$$PI(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_{n/2-1} x^{n/2-1}$$
$$PD(x) = p_{n/2} + p_{n/2+1} x + p_{n/2+2} x^2 + \ldots + p_{n-1} x^{n/2-1}$$

Q(x):

$$QI(x) = q_0 + q_1 x + q_2 x^2 + \ldots + q_{n/2-1} x^{n/2-1}$$
$$QD(x) = q_{n/2} + q_{n/2+1} x + q_{n/2+2} x^2 + \ldots + q_{n-1} x^{n/2-1}$$

Entonces:

$$P(x) = PI(x) + PD(x)\, x^{n/2}$$
$$Q(x) = QI(x) + QD(x)\, x^{n/2}$$

y el algoritmo Divide y Vencerás nos llevaría a:

$$P(x)\cdot Q(x) = PI\,(x)QI\,(x) + (PI\,(x)QD\,(x) + PD\,(x)QI\,(x))\, x^{n/2} + PD\,(x)QD\,(x)\, x^{n}$$

El algoritmo subyacente es de orden cuadrado, como el convencional que se desprende de la multiplicación de polinomios
Sin embargo, con el mismo método que en los problemas anteriores, la multiplicación de los polinomios puede hacerse teniendo en cuenta que

$$RI\,(x) = PI\,(x)\,QI\,(x)$$
$$RD\,(x) = PD\,(x)\,QD\,(x)$$
$$RH\,(x) = (PI\,(x) + PD\,(x))\,(QI\,(x) + QD\,(x))$$

Entonces la multiplicación de P por Q queda:

$$P(x)\cdot Q\,(x) = RI\,(x) + (RH\,(x) - RI\,(x) - RD\,(x))\, x^{n/2} + RD\,(x)\, x^{n}$$

Que conlleva "solo" tres multiplicaciones de polinomios de grado mitad que los originales, con la consiguiente rebaja en tiempo.
Por ejemplo, si

$$P\,(x) = 1 + x + 3x^2 + 4x^3$$
$$Q\,(x) = 1 + 2x - 5x^2 - 3x^3$$

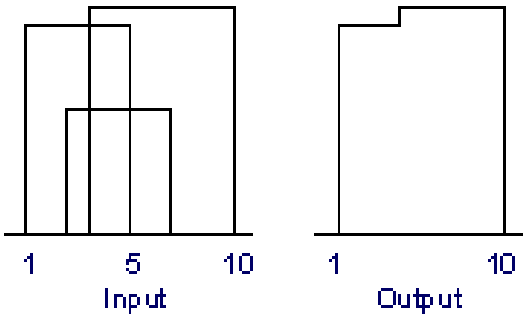
Entonces,

$$RI\,(x) = (1 + x)\,(1 + 2x) = 1 + 3x + 2x^2$$
$$RD(x) = (3 - 4x)\,(-5 - 3x) = -15 + 11x + 12x^2$$
$$RH(x) = (4 - 3x)\,(-4 - x) = -16 + 8x + 3x^2$$

y solo queda hacer las multiplicaciones.

## El problema de la línea del horizonte

Con la comercialización y popularización de las estaciones de trabajo graficas de alta velocidad, el CAD ("computer-aided design") y otras áreas (CAM, diseño VLSI) hacen un uso masivo y efectivo de los computadores. Un importante problema a la hora de dibujar imágenes con computadores es la eliminación de líneas ocultas, es decir, la eliminación de líneas que quedan ocultadas por otras partes del dibujo. Este interesante problema recibe el nombre de Problema de la Línea del Horizonte ("Skyline Problem").
El problema se establece en los siguientes sencillos términos: Dadas las situaciones exactas (coordenadas) -por ejemplo (1,11,5), (2.5,6,7), (2.8,13,10), donde cada valor es la coordenada izquierda, la altura y la coordenada derecha de un edificio- y las formas de n edificios rectangulares en una ciudad bi-dimensional,




construir un algoritmo Divide y Vencerás que calcule eficientemente el "skyline" (en dos dimensiones) de esos edificios, eliminando las líneas ocultas.
Para la solución de este problema, recurriremos a las siguientes definiciones,
**1. Tamaño:** el problema es una colección de edificios. Así el número de edificios en una colección es una elección natural para medir el tamaño del problema,

Tamaño(Edificios) = número de edificios en el input.

**2. Caso base del problema:** Como el input está restringido a la colección de edificios, entonces la base del problema es una colección constituida por un solo edificio, puesto que el "skyline" de un único edificio es el mismo edificio. Por tanto

TamañoCasoBase = 1

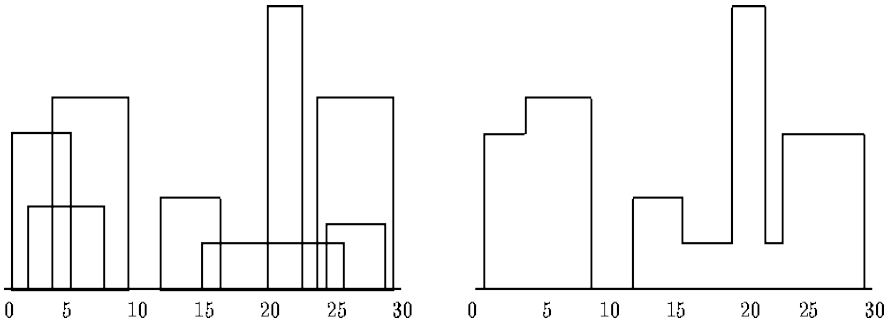
**3. Solución del Caso Base**: Como la solución para un único edificio es el edificio en sí mismo, el cuerpo del procedimiento

EncuentraSolucionCasoBase(Edificios; Skyline)

puede definirse como sigue:

```
Procedimiento EncuentraSolucionCasoBase(Edificios; Skyline)
              Begin
                 Skyline = Edificios[1]
              end;
```

**4. División del problema**: El único requerimiento es que el tamaño de los subproblemas obtenidos, de la colección de edificios, debe ser estrictamente menor que el tamaño del problema original. Teniendo en cuenta la complejidad en tiempo del algoritmo, funcionará mejor si los tamaños de los subproblemas son parecidos.
Diseñaremos un procedimiento de división de manera que los dos subproblemas que obtengamos, EdificiosIzquierda y EdificiosDerecha, tengan tamaños lo más parecidos posible.
**5. Combinación**: Hay que combinar dos skylines subsoluciones en un único skyline. La parte de combinación del algoritmo lo que hace es tomar dos skylines, analizarlos punto a punto de izquierda a derecha, y en cada punto toma el mayor valor de los dos. Ese punto es el que toma como valor del skyline combinado
El algoritmo se entiende mucho mejor si se desarrolla numéricamente sobre algún caso práctico. Consideramos el siguiente "skyline"




Entonces,

**Input (skyline de la izquierda)**

(1,11,5), (2,6,7), (3,13,9), (12,7,16), (14,3,25), (19,18,22), (23,13,29), (24,4,28)

**Output (skyline de la derecha)**

1 11 3 13 9 0 12 7 16 3 19 18 22 3 23 13 29 0

Finalmente, el algoritmo Divide y Vencerás del “skyline” es el siguiente,

```
PROCEDIMIENTO SKYLINE(Edificios,Skyline)
  Begin
    Si CasoBase(Edificios)
     Entonces
      EncuentraSolucionCasoBase(Edificios,Skyline)
      Si no
      Begin
        Dividir(Edificios, EdificiosIzquierda, EdificiosDerecha)
        EncuentraSkyline(EdificiosIzquierda, SkylineIzquierda)
        EncuentraSkyline(EdificiosDerecha, SkylineDerecha)
        Combina(SkylineIzquierda, SkylineDerecha, Skyline)
      End
  End;
```