AA2021a-04-Analisis-Algoritmos 3

Abdiel E. Cáceres González

20 de febrero de 2021

Índice

Código de Honor

La UJAT espera que sus estudiantes muestren respeto por el orden, la moral y el honor de sus compañeros, sus maestros y su persona. Por ello, se establece este Código de Honor, con el propósito de guiar la vida escolar de los alumnos. Estos lineamientos no permiten todo acto que deshonre los ámbitos académicos. Las siguientes acciones son consideradas como violatorias al Código de Honor de:

1. Usar, proporcionar o recibir apoyo o ayuda no autorizada al presentar exámenes y al elaborar o presentar reportes, tareas y en general en cualquier otra forma por la que el maestro evalúe el desempeño académico del estudiante.

2. Presentar trabajos o exámenes por otra persona, o permitir a otro que lo haga por uno mismo.

Las sanciones podrán ser desde la reprobación con calificación 0 (cero) en la tarea por evaluar, hasta una calificación reprobatoria de 0 (cero) en la materia.

PARTE DOS

Análisis de algoritmos

4. Recurrencias

4.1. Estrategia divide y vencerás

Un algoritmo diseñado bajo la estrategia de **divide y vencerás** proporciona un medio sencillo y poderoso de resolver un problema. En un algoritmo de tipo divide y vencerás, no es claro cuántas veces se realiza una tarea, porque eso depende de las llamadas recursivas y quizá de algún procedimiento previo o posterior. Los algoritmos divide y vencerás trabajan de acuerdo al siguiente procedimiento:

1. **Dividir:** Un problema se divide en dos o más subproblemas del mismo tipo, idealmente del mismo tamaño.
2. **Vencer:** Los subproblemas se resuelven [típicamente de forma recursiva, en otras ocasiones se utiliza un algoritmo diferente, especialmente cuando los subproblemas son suficientemente pequeños].
3. **Combinar:** Si es necesario, las soluciones a los subproblemas se combinan para obtener una solución al problema original.

En la figura 1 se muestra un esquema acerca del proceso que se lleva a cabo en un algoritmo de tipo divide y vencerás . En la imagen se ha dividido el problema en dos partes iguales, pero no es una regla; cada subproblema puede ser resuelto utilizando un esquema divide y vencerás o bien utilizar otro algoritmo si la entrada es suficientemente pequeña como para ser resuelto en tiempo constante.

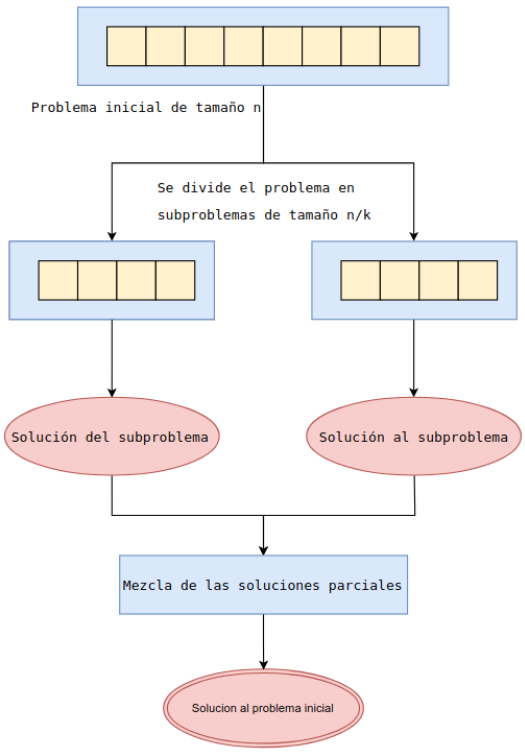

Figura 1: Esquema del proceso que sigue un algoritmo de tipo «divide-y-vencerás».

Supongamos el problema de calcular la suma de n números. Los números están alojados en un arreglo $A \leftarrowtail \langle a_0, a_1, \ldots, a_{n-1} \rangle$, así cada entrada del arreglo A es $a_0, a_1, \ldots, a_{n-1}$, ya que en muchos lenguajes de programación los índices para los arreglos se basan en el índice 0 como el primero del arreglo. Ahora, si $n > 1$, podemos dividir el problema en dos partes, cada una de ellas debe ser resuelta con la suma de los números que tiene cada parte del arreglo, de modo que la solución se puede expresar como:

> Para encontrar la suma de los números $a_0, a_1, \ldots, a_{n-1}$ hay que sumar $\langle a_0 + \cdots + a_{\lfloor \frac{n}{2} \rfloor - 1} \rangle$, con la suma en $\langle a_{\lfloor \frac{n}{2} \rfloor} + \cdots + a_{n-1} \rangle$.
>
> Al repetir consecutivamente el proceso con cada subproblema, eventualmente el subproblema llega a ser suficientemente pequeño como para resolverlo rápidamente por algún otro medio. En el ejemplo, cuando el subproblema es de tamaño 1, el resultado es inmediato, se devuelve a_0.
>
> Una vez resueltos todos los subproblemas, se procede a la parte del procedimiento en que se deben mezclar los resultados para alcanzar el resultado al problema inicial.

Este tipo de algoritmos se utilizan con frecuencia en problemas que involucran procesamiento en paralelo, donde cada hilo de operación debe resolver un subproblema.

Puede ocurrir el caso en que los problemas son suficientemente grandes de modo que se requiere resolverlos de manera recursiva, llamamos a este el **caso recursivo**, y cuando el problema es suficientmente pequeño de modo que no es necesario hacer procedimientos recursivos, estamos en un **caso base**. A veces además de los subproblemas que son instancias más pequeñas de un problema original, tenemos que resolver los subproblemas que no siempre son de la misma forma que el problema original. De modo que se sonsidera resolver los casos base de otro modo como parte del paso de combinar los resultados.

Para calcular el tiempo de ejecución de un algoritmo basado en el paradigma de divide y vencerás, frecuentemente se utilizan ecuaciones de recurrencia, o brevemene **recurrencias**. Las recurrencias van mano a mano con el paradigma divide y vencerás porque es una manera *natural* de representar el tiempo de ejecución de este tipo de algoritmos.

Así una recurrencia es una ecuación o desigualdad que describe una función en términos de los valores de las entradas más pequeñas.

4.1.1. Ordenación por mezcla [MergeSort]

Estudiemos el caso del algoritmo de ordenamiento `MergeSort`, que recibe como entrada una lista `A` de números enteros. `MergeSort` se basa en la mezcla de las soluciones, ese proceso se realiza en un algoritmo llamado `Merge` que toma dos listas de números y las mezcla, para devolver un nuevo arreglo que contiene los números ordenados.

```
PROBLEMA: mezclar la solución de dos arreglos en uno sólo
ENTRADA : A ; una lista de números
          B ; una lista de números
SALIDA  : Un arreglo que contiene los elementos de ambos, pero ordenados
          de menor a mayor.
```

```
1  Merge(A, B):
2      resultado ←↼ []
3      i ←↼ 0
4      j ←↼ 0
5      while i < |A| and j < |B|:
6          if A[i] < B[j] entonces
7              resultado ←↼ resultado + A[i] # se agrega un nuevo elemento
8              i ←↼ i + 1 # se actualiza el indice de A
9          else
10             resultado ←↼ resultado + B[j]
11             j ←↼ j + 1
12     while i < |A|:
13         resultado ←↼ resultado + A[i]
14         i ←↼ i + 1
15     while j < |B|:
16         resultado ←↼ resultado + B[j]
17         j ←↼ j + 1
18     return resultado
```

Este algoritmo `Merge` es clave para calcular el tiempo de ejecución de `MergeSort`. El tiempo de ejecución de `Merge` es $\mathbf{\Theta}(n)$, donde el valor de $n = |A| + |B|$, y A, B son los subarreglos de la lista original del problema.

El primer ciclo `while` se realiza mín$\{|A|, |B|\}$ veces, ya que termina cuando uno de los índices, ya sea `i` o bien `j` alcanza el valor de $|A|$ o de $|B|$ respectivamente. Como alguno de los índices `i` o `j` ya ha alzanzado su máximo valor, uno de los dos siguientes ciclos `while` se hace y el otro no. El ciclo `while` que sí se hace, incluye los elementos que no fueron considerados en el primer ciclo.

Ahora se puede utilizar `Merge` como una subrutima de `MergeSort`:

```
PROBLEMA : OrdenaR un arreglo de números utilizando MergeSort
ENTRADA : A ; un arreglo de números
          p ; número entero positivo
          q ; número entero positivo
SALIDA  : Un arreglo que contiene los elementos de A, pero ordenados
          de menor a mayor.
```

```
MergeSort(A: lista):
if |A| < 2 then # se detecta el caso base
  return A
else
  puntoMedio ⟵ ⌈|A|/2⌉ # se determina el punto de corte
  ladoIzq ⟵ MergeSort(A[0 ... puntoMedio]) # se divide el problema
  ladoDer ⟵ MergeSort(A[puntoMedio ... |A| − 1)]) ;
  return Merge(ladoIzq, ladoDer) # Se construye la solución
```

Cuando un algoritmo contiene al menos una llamada recursiva a sí mismo, podemos describir su tiempo de ejecución por medio de una recurrencia. Una vez que se tiene la recurrencia se pueden utilizar herramientas matemáticas para proporcionar las cotas en el desempeño del algoritmo.

4.1.2. Análisis del algoritmo `MergeSort`

Aunque los algoritmos `Merge` y `MergeSort` trabajan adecuadamente tanto en el caso cuando n es par o n es impar, para hacer más simple el análisis vamos a suponer que n es una potencia de 2. Cada paso en donde se divide el problema en dos subsecuencias de tamaño exactamente $\frac{n}{2}$. Un poco más adelante (Teorema maestro) se muestra que esta suposición no afecta el orden de crecimiento de la solución de recurrencia.

El algoritmo `Merge` con un arreglo de solamente 1 elemento, toma tiempo constante. Cuando tenemos $n > 1$ elementos, dividimos el tiempo de ejecución como sigue:

> **Dividir**: El paso de dividir el problema en subproblemas solamente calcula la mitad del subarreglo, que toma un tiempo constante, esto significa $\mathtt{D}(n) = \mathbf{\Theta}(n)$.
>
> **Vencer**: Recursivamente resolvemos cada uno de los dos subproblemas, cada uno de ellos de tamaño $\frac{n}{2}$, que contribuye $2\mathtt{T}\left(\frac{n}{2}\right)$ al tiempo total de la ejecución.
>
> **Combinar**: El procedimiento de combinar [como ya se ha comentado] toma un tiempo $\mathbf{\Theta}(n)$, de este modo $\mathtt{C}(n) = \mathbf{\Theta}(n)$.

Como antes, $\mathtt{T}(n)$ representa el tiempo de ejecución del algoritmo en un problema de tamaño n. Si n es suficientemente pequeño, digamos $n \leq c$ para algún número constante c, la solución es directa y consume un tiempo constante, que se representa como $\mathbf{\Theta}(1)$. Supongamos que la división de nuestro problema de ordenamiento genera a nuevos subproblemas, cada uno de tamaño $\frac{1}{b}$ del tamaño original, para el caso de `Merge`, $a = 2$ y $b = 2$ porque se obtienen 2 nuevos subproblemas, cada uno de tamaño $\frac{1}{2}$ del tamaño del problema original. Esto consume un tiempo de $a\mathtt{T}\left(\frac{n}{b}\right)$ para resolver a de ellos. Digamos ahora que $\mathtt{D}(n)$ es el tiempo requerido para dividir el problema en subproblemas, y $\mathtt{C}(n)$ el tiempo requerido para la combinación de las soluciones de los subproblemas en la solución del problema original, entonces obtenemos la recurrencia:

$$\mathtt{T}(n) = \begin{cases} \text{si } n \leq c & ,\mathbf{\Theta}(1) \\ \\ \text{en otro caso} & ,a\mathtt{T}\left(\frac{n}{b}\right) + \mathtt{D}(n) + \mathtt{C}(n) \end{cases}$$

Entonces al considerar todos los pasos, el tiempo de ejecución del algoritmo está dado por la recurrencia:

$$\mathtt{T}(n) = \begin{cases} \text{si } n = 1 \quad , \boldsymbol{\Theta}(1) \\ \\ \text{si } n > 1 \quad , 2\mathtt{T}\left(\frac{n}{2}\right) + \boldsymbol{\Theta}(n) \end{cases} \tag{1}$$

Al utilizar el teorema maestro [descrito más adelante], se obtiene que la solución a la recurrencia de `MergeSort` es $\mathtt{T}(n) = \boldsymbol{\Theta}(n \lg n)$, pero en realidad no es necesario el teorema maestro para determinar esta recurrencia. Escribamos la recurrencia como:

$$\mathtt{T}(n) = \begin{cases} \text{si } n = 1 \quad , c \\ \\ \text{si } n > 1 \quad , 2\mathtt{T}\left(\frac{n}{2}\right) + cn, \end{cases}$$

donde la constante c representa el tiempo requerido para resolver problemas de tamaño 1 así como el tiempo por lista de elementos en los pasos dividir y combinar.

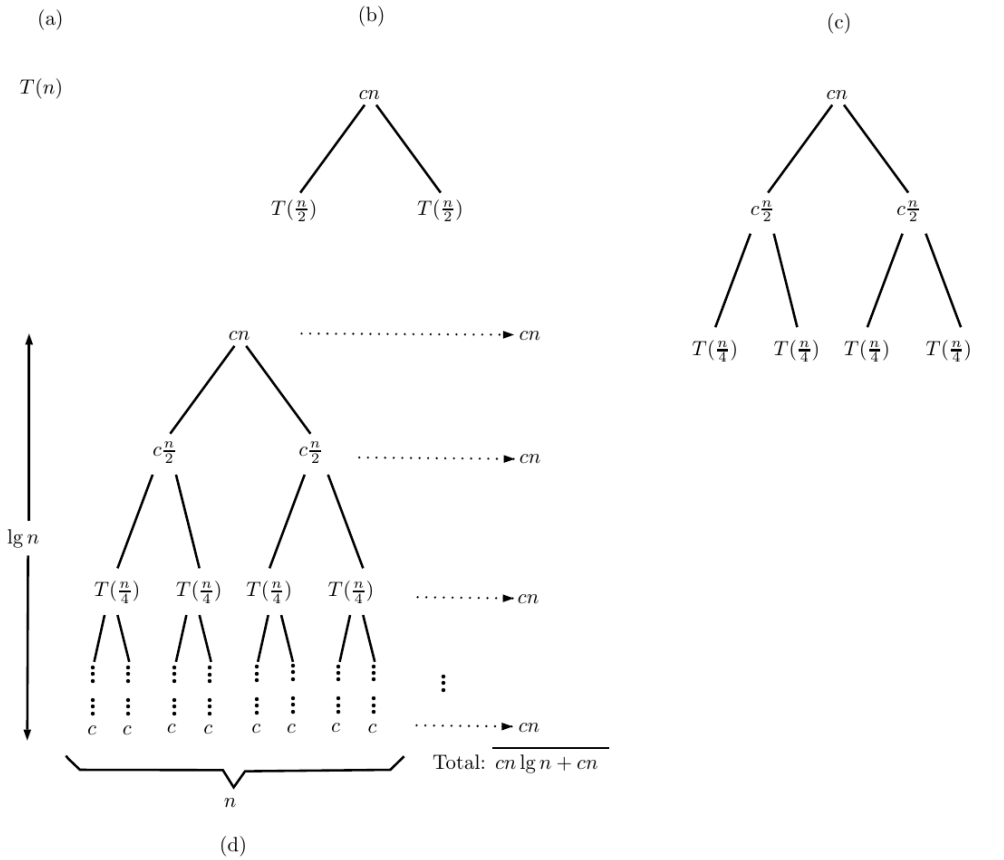

Figura 2: Construcción de un árbol de recursión para la recurrencia del `Merge` $\mathtt{T}(n) = 2\mathtt{T}(n/2) + cn$. La parte (a) muestra $\mathtt{T}(n)$, que progresivamente se expande en (b) hasta (d) para formar el árbol de recursión- El árbol todo expandido se muestra en (d) y tiene $\lg n + 1$ niveles, y cada nivel contribuye al costo total cn. Por eso el costo es $cn \lg n + cn$ que es $\boldsymbol{\Theta}(n \lg n)$.

En la figura 2 se muestra cómo podemos construir un diagrama de árbol que muestra las divisiones realizadas en el algoritmo, este es un ejemplo de un árbol de recursión.

Por conveniencia hemos supuesto que n es una potencia de 2. La parte (a) de la figura muestra el tiempo de ejecución que se debe obtener $\mathtt{T}(n)$, que se expande un nivel y se muestra en (b), cada rama del árbol muestra el tiempo de ejecución para cada uno de los dos subarreglos de tamaño $n/2$. El término cn es la raíz [que es el costo de la ejecución en en nivel más alto de la recursión], y los dos subárboles de la raíz son dos recurrencias más

pequeñas $\mathtt{T}(n/2)$. La parte (c) muestra este proceso extendido un nivel más, expandiendo los subárboles de cada subproblema $\mathtt{T}(n/2)$. El costo incurrido en cada uno de los dos vértices del segundo nivel de recursión es $c\frac{n}{2}$. Continuamos expandiendo cada nodo en el árbol dividiendo cada problema actual en dos subproblemas, como lo determina la recurrencia, hasta que los subproblemas llegan a ser de tamaño 1, cada uno con un costo de c. La parte (d) muestra el árbol de recursión.

Luego sumamos los costos en cada nivel del árbol. El nivel más alto tiene un costo total de cn, el siguiente nivel tiene un costo total $c(n/2)+c(n/2) = 2c\frac{n}{2} = cn$. El siguiente nivel tiene un costo $c(n/4)+c(n/4)+c(n/4)+c(n/4) = cn$, y así en adelante. En general, el i-ésimo nivel tiene 2^i vértices, cada uno de ellos etiquetado con un costo de $2^i c\frac{n}{2^i}$. El nivel más bajo tiene n vértices que representan problemas de tamaño 1, que se hacen en un tiempo constante c, por lo que contribuye cn al tiempo total.

El número total de niveles en el árbol de recursión es $\lg n+1$, donde n es el número de hojas, que corresponde también al tamaño de la entrada. Un argumento inductivo informal justifica esta aseveración. El caso base ocurre cuando $n = 1$, donde el árbol solamente tiene hojas. Como $\lg 1 = 0$, tenemos que $\lg n + 1$ nos da el número de niveles. Ahora supongamos como hipótesis inductiva que el número de hojas de un árbol de recursión con 2^i niveles es de $\lg 2^i+1 = i+1$ porque para cualquier valor de i, tenemos que $\lg 2^i = i$. Debido a que estamos suponiendo que el tamaño de la entrada es una potencia de 2, el tamaño siguiente a considerar es 2^{i+1}. Un árbol con $n = 2^{i+1}$ hojas tiene un nivel más que el árbol con 2^i hojas, y de ese modo el número total de hojas es $(i+1)+1 = \lg 2^{i+1}+1$.

Para calcular el costo total representado por la recurrencia 1 (página 5), simplemente sumamos los costos de todos los niveles. El árbol de recursión tienen $\lg n + 1$ niveles, cada uno tiene un costo de cn, por lo que el costo total es de $cn(\lg n + 1) = cn\lg n + cn$. Ignorando los términos de orden más bajo y la constante c, el resultado es $\boldsymbol{\Theta}(n\lg n)$.

4.2. El teorema maestro para algoritmos divide y vencierás

Todos los algoritmos de tipo divide y vencerás dividen el problema en dos o más subproblemas, cada uno de los cuales es parte del problema original y luego desarrollan algún tipo de trabajo adicional para calcular la respuesta final. Como ejemplo, el algoritmo `MergeSort` opera con dos subproblemas, cada uno de ellos de la mitad del tamaño del original y luego realiza un trabajo adicional del orden $\boldsymbol{O}(n)$ para hacer la mezcla, el tiempo de ejecución toma la forma de la ecuación:

$$\mathtt{T}(n) = 2\mathtt{T}\left(\frac{n}{2}\right) + \boldsymbol{O}(n)$$

El siguiente teorema se puede utilizar para determinar el tiempo de ejecución de algoritmos de tipo divide y vencerás. Lo primero que hay que hacer es tratar de encontrar la relación de recurrencia del problema. Si la recurrencia es de una de las formas que se muestran enseguida, podemos proporcionar directamente la respuesta.

Una versión simplificada del teorema maestro, resume todos los posibles casos de un algoritmo divide y vencerás en tres casos, que varían de acuerdo a cómo es el tiempo de ejecución de la parte de juntar y resolver, comparado con la parte de dividir el problema en subproblemas.

Teorema maestro

Si la recurrencia es de la forma $\mathtt{T}(n) = a\mathtt{T}\left(\frac{n}{b}\right)+\boldsymbol{O}(n^c)$, donde $a \geq 1$, $b > 1$ y $c \geq 0$, entonces:

Caso 1: Si $\log_b a < c$, entonces $\mathtt{T}(n) = \boldsymbol{O}(n^c)$

Caso 2: Si $\log_b a = c$, entonces $\boldsymbol{O}(n^c \log_b n) = \boldsymbol{O}(n^c \log n)$

Caso 3: Si $\log_b a > c$, entonces $\boldsymbol{O}\left(n^{\log_b a}\right)$

Estudiemos el teorema maestro con algunos ejemplos.

1. El algoritmo `MergeSort` tiene una ecuación de recurrencia $2\mathtt{T}\left(\frac{n}{2}\right) + \boldsymbol{O}(n)$ [la notación $\boldsymbol{\Omega}(\bullet)$ tiene una cota superior de $\boldsymbol{O}(\bullet)$]. Observamos que:

   $a = 2$ es el número de divisiones del problema original.

   $b = 2$ es el divisor del tamaño original, en este caso cada subparte del problema es de la mitad [es decir $\frac{1}{b=2}$] del tamaño original.

   $c = 1$ es la potencia de n en la función $\boldsymbol{O}(n^c)$, que representa la complejidad algorítmica de la parte del algoritmo que corresponde al procedimiento de mezclar y resolver.

   Como $\log_2 2 = 1$, esto nos conduce al caso 1 del teorema maestro, por lo que la complejidad del algoritmo es:

$$\begin{aligned}\boldsymbol{O}(n^c \log_b n) &= \boldsymbol{O}\left(n^1 \log_2 n\right)\\ &= \boldsymbol{O}(n \log_2 n)\\ &= \boldsymbol{O}(n \log n)\end{aligned}$$

2. El algoritmo de búsqueda binaria tiene una ecuación de recurrencia de la forma $\mathtt{T}(n) = \mathtt{T}\left(\frac{n}{2}\right) + \boldsymbol{O}(1)$, ya que al dividir la lista de números, el proceso continúa únicamente en la parte en donde se encuentra el número buscado, y la parte del algoritmo que se encarga de hacer la división es de una complejidad constante, de modo que:

   $a = 1$ significa que el proceso recursivo se hace solamente en una de las subdivisiones de la lista original.

   $b = 2$ Significa que cada subdivisión es un medio de la lista original.

   $c = 0$ porque $n^0 = 1$, que es el exponente de n [el tamaño del problema] para que la complejidad sea 1.

   Como $\log_2 1 = 0$ y $c = 0$, tomamos el caso 2:

$$\begin{aligned}\boldsymbol{O}(n^c \log_b n) &= \boldsymbol{O}\left(n^0 \log_2 n\right)\\ &= \boldsymbol{O}(1 \cdot \log_2 n)\\ &= \boldsymbol{O}(\log n)\end{aligned}$$

El A. de búsqueda binaria sirve para determinar si un número se encuentra en una lista ordenada de números, dando como entrada una lista ordenada en forma creciente y un número a buscar. La salida es `True` si el número se encuentra o `False` si no se encuentra.

4.3. Multiplicación de Karatsuba

En 1969 fue dado a conocer un algoritmo divide y vencerás que resuelve la multiplicación de dos números en un tiempo del orden $\boldsymbol{\Theta}\left(n^{\log_2 3}\right)$, que es un tiempo menor que $\boldsymbol{\Omega}\left(n^2\right)$ que es el tiempo de ejecución que se creía era la cota inferior para una multiplicación de esta naturaleza.

El algoritmo fue dado a conocer por Karatsuba, que en ese entonces tenía 25 años de edad. Para describir el procedimiento iremos construyendo un ejemplo.

Supongamos que se desean multiplicar dos números de n cifras, digamos $x \leftarrow\!\!\shortmid 5678$ y $y \leftarrow\!\!\shortmid 1234$, por lo que se desea encontrar el producto 5678×1234 por el método Karatsuba.

Antes de continuar, recordemos el método convencional de hacer este producto, le llamaremos el método de primaria [porque es en la primaria donde se aprende a multiplicar de ese modo].

$$\begin{array}{rr} & 5678 \\ \times & 1234 \\ \hline & 22712 \\ & 17034\_ \\ & 11356\_\_ \\ & 5678\_\_\_ \\ \hline & 7006652 \end{array}$$

En el método de la primaria hay que hacer del orden de n^2 multiplicaciones sencillas, pero en el método Karatsuba se busca resolver el problema con menos multiplicaciones.

Vamos a establecer en primer lugar una notación básica. El número x que es de $n = 4$ cifras, se puede escribir al concatenar 56 con 78, es decir, que $x = (56 \times 10^{\frac{n}{2}}) + 78$. La parte izquiera la denoratemos a y la parte derecha b. Por su parte, el número y que también es de $n = 4$ cifras, se puede escribir al concatenar 12 con 34, es decir, que $y = (12 \times 10^{\frac{n}{2}}) + 34$. La parte izquiera la denoratemos c y la parte derecha d.

Así los números x y y se pueden escribir:

$$x = \boxed{a = 56}\boxed{b = 78}$$

$$y = \boxed{c = 12}\boxed{d = 34}$$

Ahora procedemos en 5 pasos:

$p1$: Encontrar el producto $a \cdot c$

$p2$: Encontrar el producto $b \cdot d$

$p3$: Calcular el producto de las sumas $(a + b) \cdot (c + d)$

$p4$: Calcular $p3 - p2 - p1$

$p5$: Hacer la suma:

$$p1 \times (10^{[p2]}) + p2 + (p4 \times 10^{\frac{[p2]}{2}}),$$

donde $[p2]$ es la cantidad de cifras que tiene el número representado por $p2$.

Veamos en nuestro ejemplo:

$p1$: $ac = 56 \cdot 78 = 672$

$p2$: $bd = 12 \cdot 34 = 2652$

$p3$: $(a + b) \cdot (c + d) = 134 \cdot 46 = 6164$

$p4$: $6164 - 2652 - 672 = 2840$

$p5$: $(6164 \times 10^4) + 2652 + (2840 \times 10^2) = 6720000 + 2652 + 284000 = 7006652$

Ahora vamos a describir el algoritmo de un modo más conciso:

Primero escribimos los números x y y como:

$$x = 10^{\frac{n}{2}} a + b$$

$$y = 10^{\frac{n}{2}} c + d,$$

con a, b, c, d números de $\frac{n}{2}$ dígitos. Luego

$$\begin{aligned} x.y &= (10^{\frac{n}{2}} a + b) \cdot (10^{\frac{n}{2}} c + d) \\ &= 10^n ac + 10^{\frac{n}{2}} (ad + bc) + bd \end{aligned}$$

La idea es que de forma recursiva se calculen los productos ac, ad, bc y bd, luego agregar 0's a la derecha a las cantidades multiplicadas por una potencia de 10, de la forma que mejor convenga y finalmente hacer la suma de los términos.

Sin embargo estas cuatro multiplicaciones pueden reducirse a tres, empleando una observación que se atribuye a Gauss, quien notó esta mejora al ultiplicar números complejos[1]. Observa que al multiplicar

$$(a+b)\cdot(c+d) = ac + ad + bc + bd$$

Así que para tener el término $(ad+bc)$, habrá que restar

$$ac + ad + bc + bd - ac - bd = ad + bc$$

mi punto de vista:
(am + b)x(cm + d) =
acm^2 + (ad + bc) x m + bd

Lo bueno es que ac y bd ya han sido calculados en los primeros pasos del algoritmo.

Nota importante: No hay que olvidarse de que en un programa recursivo se debe considerar el caso base. El caso base en esta multiplicación es cuando los términos a ser multiplicados son de un solo dígito, donde se puede hacer la multiplicación convencional.

4.4. Decrementa y vence

La técnica de decrementar para resolver problemas se basa en explotar la relación entre una solución a una instancia de un problema y una solución a su instancia más pequeña. Una vez que se ha establecido una relación, se peude explotar ya sea de arriba hacia abajo [desde la instancia más compleja a la más simple], o de abajo hacia arriba [de la instancia más simple a la más compleja]. El proceso aclama una implementación recursiva por naturaleza. Esta técnica se conoce también como **modo decremental**.

Hay tres variaciones principales de esta técnica:

1. Decrementar por una constante.
2. Decrementar por un factor constante.
3. Decrementar por un factor variable.

4.4.1. Decremento constante

En la variación de **decremento constante**, el tamaño de una isntancia se reduce por la misma constante en cada aplicación del algoritmo. Típicamente la constante es igual a 1, o a una unidad, aunque nada impide otros tamaños de reducciones, de acuerdo al problema a resolver.

Para ilustrar las principales variaciones de la técnica de decrementar y vencer, estudiaremos el problema de elevar un número a a una potencia n, supondremos que $n \in \mathbb{Z}^{\geq 0}$.

En la variación decremento por una constante, el problema de la exponenciación se puede resolver mediante un algoritmo que lleve la cuenta del exponente, desde n hasta 0, considerando que cualquier número elevado a la potencia 0 es 1. En el siguiente algoritmo hemos creado una función llamada `expt`, que recibe como argumentos de entrada el número base a y el exponente n al que se desea elevar a; el objetivo es construir una solución efectiva para a^n, cuando $n \in \mathbb{Z}^{\geq 0}$.

$$\texttt{expt}(a,n) = \begin{cases} \text{si } n = 0 & ,1 \\ \text{si } n > 0 & ,a * \texttt{expt}(a, n-1) \end{cases}$$

Observa que en primera instancia se encuentra el caso cuando el exponente es mínimo, es decir cuando $n = 0$. En la notación convencional de matemáticas se puede elegir la acción en dependencia del cumplimiento de la condición expresada a la derecha, y la diferencia entre la notación convencional y la interpretación computacional es que cada

[1] Sobre el tema del famoso «truco de Gauss» hay un buen comentario en https://mathoverflow.net/questions/319559/gauss-trick-vs-karatsuba-multiplication.

expresión es evaluada de arriba hacia abajo, por lo que es importante el orden en la evaluación de las expresiones.

Observa ahora que en la llamada recursiva el decremento del exponente es en una unidad, pero así mismo cada vez que el exponente es decrementado hay un producto de a por lo que resulte de la llamada recursiva.

La expresión anterior se puede expresar en términos algorítmicos como:

Código fuente 1: n-ésima potencia de un número a. Método costoso

```
1  def expt(a, n):
2    """
3    Parameters
4    ----------
5    a : NUMERO
6       Nmero .
7    n : Nmero  entero positivo
8      El exponente al que se va a elevar el nmero a.
9
10   Returns
11   -------
12   Un  nmero  entero.
13   """
14   if (n==0):
15     return 1
16   else:
17     return a * expt(a,n-1)
18
19 x,y  = [int(i) for i in input().split()]
20 print(expt(x,y))
```

Hay ahora otros elementos que tomar en consideración y que son importantes a la hora de evaluar el desempeño de los algoritmos, y que constituye una diferencia entre las soluciones descriptivas (dadas por las expresiones matemáticas) y las soluciones efectivas (dadas por las expresiones algorítmicas o programas). Consideremos por ejemplo la ejecución de la instancia `expt`$(3, 5)$, y veamos ahora los procesos que se llevan a cabo durante el desarrollo de la ejecución del algoritmo.

```
expt(3,5)
  [3 * expt(3,4)]
  [3 * [3 * expt(3,3)]]
  [3 * [3 * [3 * expt(3,2)]]]
  [3 * [3 * [3 * [3 * expt(3,1)]]]]
  [3 * [3 * [3 * [3 * [3 * expt(3,0)]]]]]
  [3 * [3 * [3 * [3 * [3 * 1]]]]]
  [3 * [3 * [3 * [3 * 3]]]]
  [3 * [3 * [3 * 9]]]
  [3 * [3 * 27]]
  [3 * 81]
243
```

Observa el patrón de crecimiento hacia la derecha y hacia abajo. El crecimiento hacia la derecha haciendo una especie de onda es el espacio en la memoria; significa que hay operaciones que no se pueden realizar y se están acumulando en la memoria de la computadora. Y no se pueden realizar porque hay una o más procedimientos que se deben realizar para obtener los valores que requiere el procedimiento para continuar la evaluación. El crecimiento hacia abajo significan invocaciones a un procedimiento, esto implica tiempo de ejecución.

Notamos que cada evaluación de `a*expt(a,n-1)` no se puede ejecutar hasta haber calculado `expt(a,n-1)`, y esto no ocurre sino hasta que `n=0`, que es el caso fundamental; a este tipo de operaciones se les llama **operaciones diferidas**. El problema con las operaciones diferidas es que se deben alojar en la memoria de la computadora hasta que el valor de todos los operandos esté calculado y sean disponibles para la operación. Mantener espacio en la memoria de la computadora es lo que hace ineficiente (en términos de espacio) el algoritmo, porque el tiempo de ejecución $T(n) = \boldsymbol{O}(2n)$ ya que el número de operaciones que se realizan está en función del exponente, al final de cuentas se realizan n multiplicaciones, aunque hay que esperar n momentos para poder empezar a hacerlas. El espacio en memoria se puede expresar de manera similar al tiempo de ejecución, con una función $S(n)$, $S(n) = \boldsymbol{O}(n)$ ya que se deben almacenar n multiplicaciones diferidas.

Es posible hacer una mejora al algoritmo anterior. La mejora del algoritmo trabaja sobre la memoria que se emplea. El costo espacial se reduce significativamente al llevar el

resultado parcial del producto en cada iteración, de modo que cuando el exponente llegue a su valor mínimo, el producto parcial contenga el valor definitivo de la exponenciación.

$$\texttt{expt}(a, n, r = 1) = \begin{cases} \text{si } n = 0 & , r \\ \\ \text{si } n > 0 & , \texttt{expt}(a, n-1, a * r) \end{cases}$$

Una primera observación es la forma $res = 1$ dentro de los argumentos de la función `expt`. Esta notación significa que el valor por defecto del parámetro res es 1, a menos que se invoque la función con otro valor.

Código fuente 2: Cálculo de la n-ésima potencia de un número a

```
def expt(a,n, res = 1):
  """
  Parameters
  ----------
  a : NUMERO
     Nmero .
  n :  Nmero  entero positivo
    El exponente al que se va a elevar el nmero a.
  res = 1 : parmetro opcional, construye el resultado
  Returns
  -------
  Un  nmero  entero.
  """
  if (n==0):
    return res
  else:
      return expt(a, n-1, res * a)

x,y  = [int(i) for i in input().split()]
print(expt(x,y))
```

Para ver la diferencia en los efectos que produce este ligero cambio en la interpretación de la solución, hagamos `expt(3,5)` con esta nueva versión.

```
expt(3,5) ; equivale a expt(3,5,1)
  expt(3,4,3)
  expt(3,3,9)
  expt(3,2,27)
  expt(3,1,81)
  expt(3,0,243)
243
```

La complejidad temporal no cambia, es decir es de orden $\boldsymbol{O}(n)$, pero la complejidad espacial ahora se reduce a ser constante $\boldsymbol{O}(1)$, porque el valor del resultado final r se almacena en la misma variable, que se actualiza en cada iteración.

4.4.2. Decremento por un factor constante

Veamos ahora una tercera versión del algoritmo de exponeciación, ahora para ejemplificar la variación decremento por un factor constante. Esta nueva versión se basa en algunas propiedades de los exponentes, específicamente

$$a^{mn} = a^m \times a^n,$$

y observando que cuando el exponente n es un número par,

$$a^n = a^{\frac{n}{2}} \times a^{\frac{n}{2}} = (a^{\frac{n}{2}})^2, \tag{2}$$

esto permite hacer menos operaciones. Observe el comportamiento con unos ejemplos:

$$\begin{aligned} b^{16} &= b^8 \times b^8 \\ b^8 &= b^4 \times b^4 \\ b^4 &= b^2 \times b^2 \\ b^2 &= b^1 \times b^1 \end{aligned}$$

Pero este truco solamente funciona cuando el exponente es una potencia de 2. En general puede ser que el exponente sea par o impar. Cuando el exponente es un número

par se puede realizar la expresión 2, y en caso de que el exponente sea impar, simplemente hacemos

$$a \times a^{n-1}$$

Así considerando los siguientes tres procedimientos, se puede enunciar un algoritmo que realiza menos multiplicaciones para determinar la n-ésima potencia de un número a.

$$\texttt{par?}(n) = \begin{cases} \text{si } n \mod 2 = 0 & ,\texttt{V} \\ \text{si } n \mod 2 = 1 & ,\texttt{F} \end{cases}$$

$$\texttt{cuad}(n) = n \times n;$$

$$\texttt{expt}(a,n) = \begin{cases} \text{si } n = 0 & ,1 \\ \text{si } \texttt{par?}(n) & ,\texttt{cuad}(\texttt{expt}(a, \frac{n}{2})) \\ \text{si } \neg\texttt{par?}(n) & ,a \times \texttt{expt}(a, n-1) \end{cases}$$

Nota que el decremento del exponente es de la mitad, cada vez que el exponente es un número par, y es de 1 cada vez que el número es impar.

El proceso llevado a cabo por el algoritmo `expt-rap` crece logaritmicamente a medida que el exponente n crece, tanto en espacio como en tiempo. Observa que calcular a^{2n} solamente requiere una multiplicación más que calcular 2^n.

La diferencia entre el crecimiento de $\mathbf{\Theta}(\log n)$ con el crecimiento de $\mathbf{\Theta}(n)$ se vuelve crítica a medida que n aumenta. Por ejemplo mientras que el algoritmo `expt` requiere 1000 multiplicaciones para calcular una potencia con $n = 1000$, con el algoritmo `expt-rap` se requieren solamente 16 (cuando n cambia desde 1000 hasta 0 en la sucesión $[1000, 500, 250, 125, 124,\ 62, 31, 30, 15, 14, 7, 6, 3, 2, 1, 0]$)

4.4.3. Decremento variable

Finalmente, para ilustrar la variación **decremento variable** del método decrementar-y-vencer vamos a estudiar el algoritmo para calcular en máximo común divisor (`MCD`).

Recuerda que si $n \neq 0$ y m son números enteros no negativos, m se puede escribir en términos de n como

$$m = qn + r,$$

para algunos números enteros q y r, y $0 \leq r < n$. Además, sólo hay una manera de hacer esto.

Si $r = 0$, de manera que m es un múltiplo de n, se escribe $n|m$, y se lee «n divide a m». Si m no es un múltiplo de n, se escribe $n \not| m$, y se lee «n no divide a m».

Si a, b y k son enteros positivos, y $k|a$ y $k|b$, entonces k es un **divisor común** de a y b. Si d es el mayor de esos k, a d se le llama **máximo común divisor** o `MCD`, y se escribe $d = \texttt{MCD}(a, b)$.

Entonces para determinar el máximo común divisor de dos números a y b se puede seguir el siguiente algoritmo:

$$\texttt{MCD}(a,b) = \begin{cases} \text{si } b = 0 & ,a \\ \text{si } b \neq 0 & ,\texttt{MCD}(b, a \bmod b). \end{cases}$$

Observa en este caso que cada vez que se invoca el procedimiento `MCD` se requiere calcular el nuevo valor del parámetros b, de modo que no es un decremento constante, tampoco es un decremento dado por un factor constante.

Ejercicios

Criterio de evaluación

Para que seas conciente de cómo se evalúan y califican estos ejercicios, considera el siguiente criterio que otorga cierto número de puntos por cada ejercicio:

- 3: Un ejercicio entregado a tiempo y sin alguna falla. Un ejercicio de programación corre a la perfección; sigue fielmente las instrucciones y su notación, y tiene su contrato correctamente escrito.
- 2: Un ejercicio mayormente correcto entregado a tiempo. Un ejercicio de programación o bien no tiene su contrato; o bien ha cambiado identificadores o es parcialmente correcto.
- 1: Un ejercicio entregado a destiempo o mayormente equivocado. Un ejercicio de programación no tiene su contrato y ha cambiado identificadores y es parcialmente correcto.
- 0: Un ejercicio completamente equivocado o no se entrega.

La calificación de la lista de ejercicios se obtiene al ponderar el número de puntos alzanzado en una escala de 0 a 10.

1. Escribe un programa en `Python` llamado `bbinaria.py` que resuelva el problema de la búsqueda binaria, siguiendo el diseño «divide y vencerás». Escribe en comentarios dentro del código las secciones «dividir», «vencer» y «combinar». Nota: Este ejercicio de programación es uno de los «clásicos», por lo que puedes encontrar la solución en Internet, sin embargo te invito a que lo trates de hacer por ti mismo, para que el logro sea completamente tuyo.

```
PROBLEMA:
Determinar si un número <n> dado está dentro de una lista <lst> ordenada de números.
ENTRADA:
lst : lista de números enteros, ordenada en forma creciente.
n : número entero.
SALIDA:
True: Si <n> está en la lista de números.
False: Si <n> no está en la lista de números.
```

```
1  BBINARIA(lst,n)
2  if (lst = [])
3    return False
4  else
5    medio ←- |lst|/2
6    A = lst[0..medio-1]
7    B = lst[medio+1 .. n]
8    caso (n = lst[medio])
9      return True
10   caso (n < lst[medio])
11     return BBINARIA(A,n)
12   else
13     return BBINARIA(B,n)
```

Observa que la entrada de datos no requiere ningún texto previo a la captura, el programa simplemente debe iniciar, el usuario debe ingresar los datos uno a uno y el programa termina mostrando un valor booleano.

Entrada	Salida
↵ 5	False
0 1 1 3 5 8 9 9 12 5	True
0 1 1 3 5 8 9 9 12 17	False

2. En el algoritmo Karatsuba de la página 7 se describe un algoritmo que sigue el modelo divide y vencerás. Describe qué partes del algoritmo corresponden a cada etapa del procedimiento, recuerda que las etapas son dividir, vencer y combinar.

3. Escribe un programa en `Python` que se llame `karatsuba.py` y que reciba dos números enteros positivos y que los multiplique utilizando el algoritmo `karatsuba`. El resultado debe ser un número entero.

```
PROBLEMA:
Calcular el producto de dos números enteros mediante el algoritmo Karatsuba.
ENTRADA:
x : número entero.
y : número entero.
SALIDA:
z : número entero.
```

Entrada	Salida
539023423 90123	48578407951029
1234 5678	7006652

4. Escribe un programa recursivo en `Python` que se llame `masvocales.py`. El programa debe tener como parámetro de entrada un texto llamado `tx` y como salida debe ofrecer un valor `True` si en el texto hay más vocales que consonantes, o incluso un número igual; debe devolver `False` si en el text hay mas consonantes que vocales.

```
PROBLEMA:
Determina de manera recursiva si en un texto hay mas consonantes que vocales
ENTRADA:
tx : Una cadena de caracteres
SALIDA:
booleano. True si el numero de vocales es mayor o igual a las consonantes. False eoc
```

Entrada	Salida
ab1	True
aei k	True
aei kktw	False

5. Escribe un programa en `Python` que se llame `expt.py` para calcular la n-ésima potencia de un número a mediante el modelo de decremento por un factor constante [página 11].

```
PROBLEMA:
Calcular pa n-esima potencia de un número entero
ENTRADA:
a : número entero
n : número entero no negativo
SALIDA:
número entero
```

Entrada	Salida
2 3	8
100 0	1

6. Escribe un programa en `Python` que siga un algoritmo de decremento constante para calcular el conjunto potencia de un conjunto.

```
PROBLEMA:
Calcular el conjunto potencia de un conjunto de números enteros
ENTRADA mínima (puedes poner más parámetros):
C : Una lista de números enteros diferentes
SALIDA:
Lista de listas
```

Entrada	Salida
⏎	[]
1 2	[[], [1], [2], [1, 2]]
1 2 3	[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

7. Diseña un algoritmo recursivo de decremento que encuentre el n-ésimo renglón del triángulo de Pascal. Escribe un programa en `Python` que se llame `nPascal` y que reciba como entrada un número entero no negativo. El programa debe devolver una lista con los números que conforman el `n`-ésimo renglón del triángulo de Pascal.

```
PROBLEMA:
Calcular el n-ésimo renglón del triángulo de Pascal
ENTRADA mínima (puedes poner más parámetros):
n : número entero
SALIDA:
Lista de números
```

Entrada	Salida
`0`	`[1]`
`1`	`[1, 1]`
`2`	`[1, 2, 1]`