Divide and Conquer

Divide and Conquer es un paradigma de diseño de algoritmos basado en la recursividad de soluciones. Se separa en tres fases:

1. ***Divide***: separar un problema en pequeñas instancias de sí mismo. Por ejemplo, para el problema de ordenar una lista de números podría separarse la lista en sublistas, de modo que la tarea se reduzca a ordenar varias listitas en lugar de una listota.

2. ***Conquer***: aplicar una solución recursiva a los subproblemas. Como la solución es recursiva, los subproblemas serán continuamente subdivididos hasta alcanzar un caso base, al que se aplica un procedimiento de solución directo.

3. ***Combine***: claramente, la recursión tendrá diferentes niveles de profundidad hasta alcanzar los casos base. En cada nivel de la recursión habrá un conjunto de problemas "llamadores" esperando solución. *Combine* se encarga de proveer estas soluciones, construyéndolas mediante la unión de soluciones a subproblemas de un nivel más profundo. Esta unión debe hacerse de forma coherente con el problema general. Retomando el ejemplo del ordenamiento de números, unimos, manteniendo el orden, las listitas ya ordenadas para formar listas un poco más grandes que se deben mantener ordenadas, y que luego se unirán a otras listas medianas para formar listas ordenadas más grandes, etc.

Un algoritmo de éstos nos presenta la facilidad de expresar su tiempo de ejecución como una **relación de recurrencia**. *E.g.*:

$$a_k = a_{k-1} + 1, k \in \mathbf{Z}^+, a_0 = 0$$

Una relación de recurrencia, en el contexto de este tema, es una igualdad o desigualdad que describe una función en términos de sus resultados sobre *inputs* previos en una secuencia predeterminada. En la relación de recurrencia estos **términos recurrentes** representan el tiempo que toma ejecutar la fase de *Conquer*, mientras que los términos no-recurrentes corresponden al *overhead* (tiempo adicional) que producen las fases de división y combinación. Podemos expresar estos *overheads* con términos separados, pero es habitual combinarlos en una única expresión con notación asintótica.

Como ejemplo tomaremos el algoritmo ***Merge Sort.*** Este es un algoritmo de ordenamiento de números que aplica las fases del paradigma DaC. El pseudocódigo del algoritmo es bastante corto:

```
Merge-Sort(A,p,r):

1. If p < r:
2.       q = ⌊(p+r)/2⌋
3.       Merge-Sort(A,p,q)
4.       Merge-Sort(A,q+1,r)
5.       Merge(A,p,q,r)
```

Donde A es la lista de números a ser ordenada y p, q son los índices que delimitan qué porción de la lista está siendo ordenada.

1. Describa lo que hace este algoritmo en cada fase de *Divide and Conquer*.

2. Si $T(n)$ es el tiempo de ejecución de *Merge Sort* sobre un arreglo de tamaño n, defina $T(n)$ como una relación de recurrencia.

Hace falta tomar en cuenta el caso base. Como el caso trivial se alcanza cuando las sublistas poseen un único elemento, dicho caso se ejecuta con un tiempo constante. Nuestra relación de recurrencia queda al final así:

$$T(n) = \begin{cases} O(1)\ cuando\ n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n)\ cuando\ n > 1 \end{cases}$$

3. Describa el algoritmo para *Merge*. ¿Qué característica debe guiar su diseño?

El algoritmo para *Merge* es el siguiente:

`Merge(`A, p, q, r`):`

```
1.  n1 = q − p + 1
2.  n2 = r − q
3.  for i=1 to n1:
4.      L[i] = A[p + i − 1]
5.  for j=1 to n2:
6.      R[i] = A[q + j]
7.  L[n1 + 1] = ∞
8.  R[n2 + 1] = ∞
9.  i=1
10. j=1
11. for k=p to r:
12.     if L[i] ≤ R[j]:
13.         A[k] = L[i]
14.         i = i+1
15.     else:
16.         A[k] = R[j]
17.         j = j+1
```

4. ¿Cuál es el tiempo de ejecución de *Merge*, en notación asintótica?

En general, las ecuaciones de recurrencia que expresan el tiempo de ejecución de algoritmos DaC pueden tener diferentes formas. El *input* de *Conquer* no necesariamente debe ser una fracción del *input* original (*e.g.*, puede ser una resta), y *Conquer* puede conllevar varias ejecuciones recursivas con coeficientes y porciones de *input* de tamaños distintos.

Aunque nuestra ecuación para *Merge Sort* es hermosa, lo único que logramos fue expresar el tiempo de ejecución del algoritmo. Todavía no tenemos algo que nos sirva para evaluar su desempeño, por lo que necesitaremos emplear un método de solución de ecuaciones de recurrencia. Veremos tres.

Árbol de recursión

El primer método que veremos es el del **árbol de recursión** porque es el más intuitivo. En un árbol de recursión, cada nodo representa el costo de solucionar un subproblema y la raíz corresponde al costo de la ejecución inicial del algoritmo sobre la entrada original. Las ramas que salen de un nodo son las ejecuciones recursivas realizadas durante la ejecución del nodo del que parten. Nosotros sumamos el costo en el que incurre cada nodo para obtener el costo total de ejecución del algoritmo. ¿Y de dónde salen los costos de los nodos? De las fases *Divide* y *Combine*.

5. Represente visualmente el tiempo de ejecución de *Merge Sort* mediante un árbol de recursión. Suponga que el tamaño del *input* es una potencia de dos. ¿Cuántos niveles tendrá el árbol? Use el árbol para expresar el tiempo de ejecución de *Merge Sort* con notación asintótica.

Como en cada nivel se divide la entrada en dos, lo que estamos haciendo es la operación inversa a la potenciación con 2. El inverso de la potenciación con 2 es el logaritmo con base 2, entonces esa es la cantidad de niveles de recursión que tendremos antes de alcanzar el caso base, en el que la entrada ya no es divisible.

El costo de las fases *Divide* y *Combine* de *Merge Sort* es $\Theta(n)$; si imponemos una cota superior podemos argumentar que, para alguna constante *c*, el tiempo de ejecución será menor o igual a cn. En el primer nivel de la recursión no se ha dividido el *input* en subproblemas, pero en el siguiente nivel separamos el *input* en dos y mandamos cada mitad a una llamada recursiva distinta. Eso es lo que expresan las dos ramas que parten de la raíz, cada una con un *input* de tamaño $\frac{n}{2}$.

Es decir, en el segundo nivel se procesarán dos mitades con “media entrada”. En el tercero cada mitad se separa en otras dos, entonces tenemos cuatro llamadas, cada una con “un cuarto de entrada”; etc. Si nos portamos riguros@s con los costos al dibujar el árbol podemos aplicar este método para obtener resultados exactos, aunque los árboles de recursión se usan regularmente de forma relajada para obtener un resultado intuitivo y luego comprobarlo con algún otro método.

UVG
UNIVERSIDAD DEL VALLE DE GUATEMALA

Substitución

El árbol de recursión para *Merge Sort* nos indicó que el tiempo de ejecución $T(n)$ es $cn * \log_2 n$, donde c es multiplicador positivo con el que acotamos el tiempo de ejecución de las fases *Divide* y *Combine*, así como también el del caso base. Vamos a analizar este caso y luego iremos liberando las suposiciones para una perspectiva más general.

6. Suponiendo que el tiempo de ejecución de *Merge Sort* fuera $\Theta(n \log_2 n)$ con alguna constante c dada, desarrolle el valor de $T\left(\frac{n}{2}\right)$ y reemplace ese término en la relación de recurrencia. ¿Este resultado coincide con su estimación? ¿Cuál es la importancia de que sí coincida?

El **método de substitución** comienza por adivinar o proponer una solución y luego emplear inducción matemática para demostrar que la solución es correcta. El procedimiento dice "usted proponga un tiempo de ejecución (o una cota) para $T(n)$ y úselo para describir el término recursivo de la relación de recurrencia. Substituya el término recursivo en $T(n)$ con esta descripción y luego desarrolle $T(n)$. Lo que obtenga tiene que coincidir con su propuesta. Finalmente, demuestre que su caso recursivo termina mediante una condición de frontera."

La inducción matemática es un proceso que determina condiciones de frontera. Para demostrar con inducción matemática que nuestra suposición funciona, la suposición debe ser aplicable a todo $m < n$, por lo que debe funcionar para $\frac{n}{2}$ o la porción del *input* que se use en las llamadas recursivas. Esa suposición es normalmente llamada "hipótesis de inducción", y mediante la hipótesis de inducción se debe comprobar que nuestra suposición funciona también cuando $m = n$. Si esto es logrado, como arriba, sólo nos queda demostrar que nuestra suposición aplica en las condiciones de frontera para que la demostración esté completa.

Lo común es demostrar que las condiciones de frontera coinciden con los casos base de la relación de recurrencia del algoritmo. En la relación de recurrencia tenemos que, para $n = 1, T(n) = O(1)$, por lo que podemos suponer que hay una constante $c > 0$ tal que $T(1) \leq c$. Nuestra condición de frontera para *Merge Sort*, en ese caso, sería $n = 1$, pero esto engendrará un problema que resolveremos un poco más adelante. Antes, veamos otro caso, más sencillo, en el que se da el mismo clavo.

7. El truco en el procedimiento anterior fue partir de una suposición informada. Imaginemos que nos piden demostrar con el método de substitución que la siguiente recurrencia es $O(n * \log_2 n)$:

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

Nos dicen, además, que $T(1) = 1$. Desarrolle la recurrencia y encuentre el problema.

¿Qué hacer? Lo importante de partir de una notación asintótica es que nos permite manipular las constantes de su definición. Recordemos que las notaciones asintóticas determinan que sus condiciones se cumplen a partir de un cierto n_0 positivo. Si sabemos que $T(1) = 1$ y que eso no cumple con la suposición, divorciemos el caso base y la condición de frontera. Si optamos por un caso base más alto, como $n = 2$, nuestra suposición exigiría que $T(2) \leq c(2) * \log_2 2 = 2c$. En la relación de recurrencia, el caso base $n = 2$ produce:

$$T(2) = 2T\left(\frac{2}{2}\right) + 2 = 2T(1) + 2 = 2 + 2 = 4$$

¿Puede ser $4 \leq 2c$? Sí, porque la única restricción que hemos impuesto a c hasta ahora es que $c > 0$ (por definición de *big-Oh*). La restricción que agregamos ahora es que $T(n) \leq cn * \log_2 n$ para todo $n \geq 2$ con $c \geq 2$. Notemos que, con esto, $T(n) \leq cn * \log_2 n - cn + n < cn * \log_2 n$. Como $T(1) = 1 = O(1)$, y toda función $O(1)$ es también $O(n * \log_2 n)$, es válido concluir que la relación de recurrencia cumple con $T(n) = O(n * \log_2 n)$ con $c \geq 2$ y $n_0 = 2$.

Al aplicar el método de substitución pueden ocurrir otros problemas. Si nuestra suposición fallara al reemplazar los términos recurrentes en la relación de recurrencia tendríamos que reforzar nuestra cota y probar con una función más específica. En ocasiones, si el fallo es provocado por una diferencia pequeña, podemos ajustar nuestra suposición de modo que la substitución funcione.

Después de todo, usar una notación asintótica como $T(n) = O(n)$ nos permite que $T(n) \leq cn$ como también que $T(n) \leq cn \pm k$ donde *k* es una constante.

8. Demuestre que si $T(n) = O(n)$ y k es un término de orden menor a $T(n)$, $T(n) \leq cn \pm k$. *Hint*: ¿Qué pasa con $T(n) \pm k$ si sabemos que $T(n) = O(n)$?

Ilustramos esto en el siguiente ejemplo:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$$

Supongamos que ésta es la relación de recurrencia que describe el tiempo de ejecución de algún algoritmo. Proponemos que $T(n) = O(n)$, entonces deseamos probar que $T(n) \leq cn$. Procedemos aplicando la hipótesis inductiva (que dice que $T(n) \leq cn$ para todo $m < n$):

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq c \left\lfloor \frac{n}{2} \right\rfloor$$

Luego reemplazamos el término recurrente en la ecuación con este resultado:

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1 \not\leq cn$$

Fail. Si decimos mejor que $T(n) \leq cn - d$ (con d constante) como propuesta veremos que:

$$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq c \left\lfloor \frac{n}{2} \right\rfloor - d$$

Entonces:

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor - d + c \left\lceil \frac{n}{2} \right\rceil - d + 1 = cn - 2d + 1 = cn - d - (d - 1) \leq cn - d$$

y el mundo vuelve a ser color de rosa (siempre y cuando $d \geq 1$). Este es el tipo de técnica que se aplicaría al emplear el método de substitución para demostrar que *Merge Sort* es $O(n * \log_2 n)$. En un inicio supondríamos que $T(n) \leq cn * \log_2 n$, donde $T(n)$ cumple con la relación de recurrencia encontrada para *Merge Sort* al principio de este documento. La hipótesis inductiva es la que permite que $T\left(\frac{n}{2}\right) \leq \frac{cn}{2} * \log_2 \frac{n}{2}$, tal como se hizo anteriormente.

Ahora recordemos que la relación de recurrencia se separó en casos y que el caso $n = 1$ fue determinado perteneciente a $O(1)$, lo que nos permite proponer que la constante $k > 0$ es tal que $0 < T(1) \leq k$. Si intentamos que la condición de frontera y el caso base sean el mismo (*i.e.*, $n = 1$) nos topamos con el mismo problema del ejemplo anterior al probar la hipótesis inductiva con: $T(1) \leq c(1) * \log_2 1 = 0$. Es decir, aunque la relación de recurrencia no tiene problema, la condición de frontera que determinamos, según la cual $T(1) = O(1)$, no se cumple con nuestra hipótesis inductiva.

Como en el ejemplo anterior, intentemos modificar nuestra suposición a $T(n) \leq cn * \log_2 n - \beta$. Entonces:

$$T\left(\frac{n}{2}\right) \leq \frac{cn}{2} * \log_2 \frac{n}{2} - \beta$$

$$T(n) \leq \frac{2cn}{2} * \log_2 \frac{n}{2} - 2\beta + cn = cn * \log_2 n - cn - 2\beta + cn = cn * \log_2 n - 2\beta$$

$$\leq \ cn * \log_2 n - \beta$$

Necesitaríamos elegir la constante β de modo que $cn * \log_2 n - 2\beta \leq \ cn * \log_2 n - \beta$, lo cual resulta fácil pues basta con cualquier $\beta > 0$. Habiendo comprobado que nuestra relación de recurrencia no tiene problema con la nueva hipótesis inductiva, probamos nuevamente la condición de frontera como el caso base:

$$T(1) \leq c(1) * \log_2 1 - \beta = -\beta$$

Como ya habíamos planteado que $\beta > 0$, esto no funciona.

9. Demuestre que *Merge Sort* es $O(n \log_2 n)$ usando el método de substitución. Tomando en cuenta el intento anterior, modifique la hipótesis de inducción original usando otro término aditivo de orden menor a $cn \log_2 n$.

Master Method

En el caso particular en el que la relación de recurrencia tiene la forma $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ para $a \geq 1$, $b > 1$ y $f(n)$ asintóticamente positiva (llamada la **Recurrencia Maestra**) podemos emplear el ***Master Method***, un método que se apoya en el ***Master Theorem***. El *Master Theorem* funciona como receta de cocina especificando los siguientes tres casos:

- Si $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ para alguna constante $\epsilon > 0$, entonces $T(n) = \Theta\left(n^{\log_b a}\right)$.
- Si $f(n) = \Theta\left(n^{\log_b a}\right)$, entonces $T(n) = \Theta\left(n^{\log_b a} * \log_b n\right)$.
- Si $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ para alguna constante $\epsilon > 0$ y, además, $a * f\left(\frac{n}{b}\right) \leq cf(n)$ para alguna constante $c < 1$ y n suficientemente grande (esta última desigualdad se llama **condición de regularidad**), entonces $T(n) = \Theta\left(f(n)\right)$.

Vamos a demostrar el *Master Theorem* suponiendo que el *input* es de tamaño b^k para algún entero $k > 1$. El árbol de recursión de la Recurrencia Maestra tiene a ramas partiendo de cada nodo y, en cada nivel i, empezando desde $i = 0$, el número de nodos es a^i y el costo de cada nodo es $f\left(\frac{n}{b^i}\right)$.

10. Demuestre que el número de hojas de este árbol de recursión es $n^{\log_b a}$.

Lo que el *Master Method* compara es la contribución de las hojas del árbol (las ejecuciones del caso base) al tiempo de ejecución, contra la contribución de los nodos internos (las ejecuciones recursivas).

El método supone, razonablemente, que el tiempo de ejecución de las hojas (que es cuando el *input* ya no se puede dividir más) es constante. Entonces, el costo de ejecución de las hojas del árbol es $\Theta\left(n^{\log_b a}\right)$, con lo que el costo total del árbol se vuelve:

$$\Theta\left(n^{\log_b a}\right) + \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right)$$

Ahora vamos a ver los casos observados por el *Master Theorem*. El primer caso contempla que el tiempo de ejecución de las hojas del árbol domina (es cota superior) sobre el costo de los nodos internos. Cuando $f(n) = O\left(n^{\log_b a-\epsilon}\right)$ tenemos:

$$\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) = \sum_{j=0}^{\log_b n-1} a^j O\left(\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right) \leq \sum_{j=0}^{\log_b n-1} a^j c\left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}$$

$$= c \sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} \Rightarrow \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \leq c \sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} \Rightarrow$$

$$\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) = O\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon}\right)$$

Desarrollando el contenido de O, podemos sacar $n^{\log_b a-\epsilon}$ porque no depende de j:

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \frac{a^j}{b^{j*(\log_b a-\epsilon)}} = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a} * b^{-\epsilon}}\right)^j$$

$$= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{a * b^{\epsilon}}{b^{\log_b a}}\right)^j = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{a * b^{\epsilon}}{a}\right)^j = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} b^{\epsilon j}$$

Esta última sumatoria es una serie geométrica:

$$\sum_{j=0}^{\log_b n-1} b^{\epsilon j} = \frac{b^{\epsilon \log_b n-1+1} - 1}{b^{\epsilon} - 1} = \frac{\left(b^{\log_b n}\right)^{\epsilon} - 1}{b^{\epsilon} - 1} = \frac{n^{\epsilon} - 1}{b^{\epsilon} - 1} = O(n^{\epsilon})$$

La notación asintótica es válida porque b y ϵ son constantes. De modo que $\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} = n^{\log_b a-\epsilon} O(n^{\epsilon}) = O\left(n^{\log_b a}\right)$. Por lo tanto, en el primer caso, el costo total de ejecución es:

$$\Theta\left(n^{\log_b a}\right) + O\left(n^{\log_b a}\right) = \Theta\left(n^{\log_b a}\right)$$

El segundo caso del *Master Theorem* es cuando $f(n) = \Theta\left(n^{\log_b a}\right)$, entonces procedemos como antes:

$$\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right)$$

Desarrollando el contenido de Θ:

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \frac{a^j}{b^{j*\log_b a}} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1^j = n^{\log_b a} \log_b n$$

$$= \Theta\left(n^{\log_b a} \log_b n\right)$$

Entonces, el costo total de ejecución es:

$$\Theta\left(n^{\log_b a}\right) + \Theta\left(n^{\log_b a} \log_b n\right) = \Theta\left(n^{\log_b a} \log_b n\right)$$

El último caso es cuando $f(n) = \Omega(n^{\log_b a+\epsilon})$, pero debemos tomar en cuenta la condición de regularidad: $af\left(\frac{n}{b}\right) \leq cf(n)$ donde $c < 1$.

11. Explique por qué se debe cumplir la condición de regularidad. ¿Qué se expresa con la condición?

Notemos que en el nivel j la condición se vuelve $a^j f\left(\frac{n}{b^j}\right) \leq c^j f(n)$, entonces podemos proceder así:

$$\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \leq \sum_{j=0}^{\log_b n-1} c^j f(n)$$

Recordemos que la c en la condición de regularidad es menor a uno, y que dicha condición se cumple "para suficientemente grandes n". Es decir que puede haber algunos n en la sumatoria para los cuales no cumple la condición de regularidad, por lo que compensamos esto con un $O(1)$:

$$g(n) = \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \leq \left(\sum_{j=0}^{\log_b n-1} c^j f(n)\right) + O(1)$$

$$= f(n)\left(\sum_{j=0}^{\log_b n-1} c^j\right) + O(1) \leq f(n)\left(\sum_{j=0}^{\infty} c^j\right) + O(1)$$

Reemplazamos el límite superior de la sumatoria para obtener una serie geométrica decreciente que nos permite proceder así (por ser $c < 1$):

$$f(n)\sum_{j=0}^{\infty} c^j + O(1) = f(n)\left(\frac{1}{1-c}\right) + O(1) = \frac{f(n)}{1-c} + O(1) = O\big(f(n)\big)$$

12. Demuestre que, además, $\sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) = \Omega\big(f(n)\big)$. ¿Por qué podemos afirmar que $\Theta\big(n^{\log_b a}\big) + \Theta\big(f(n)\big) = \Theta\big(f(n)\big)$?

El *Master Theorem* aplica también cuando el *input* no es de tamaño $n = b^k$ para algún entero $k \geq 0$. No incluiremos la demostración para estos *inputs*, pero se puede encontrar en el libro de Cormen (Cormen, Leiserson, Rivest, & Stein, 2009).

Finalizamos con algunos ejemplos de uso del *Master Method*. Recordando la recurrencia del *Merge Sort:*

$$T(n) = \begin{cases} O(1), & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

observaremos que $a = b = 2$ y $f(n) = \Theta(n)$. Luego observemos que $n^{\log_b a} = n = \Theta(n)$ entonces $f(n) = \Theta\big(n^{\log_b a}\big)$, por lo que el caso 2 aplica. Por ello: $T(n) = \Theta\big(n^{\log_b a} \log_b n\big) = \Theta(n \log_2 n)$, comprobando la conclusión que ya habíamos obtenido con los otros métodos. Veamos otra recurrencia:

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log_2 n$$

En este caso $a = 3, b = 4$ y $f(n) = n \log_2 n$. Vemos que $n^{\log_b a} = n^{\log_4 3} = n^{0.793}$. Como la n en $n \log_2 n$ tiene exponente 1 y $\log_2 n > 0$:

$$f(n) = n \log_2 n = \Omega(n) = \Omega\left(n^{\log_b a + 0.207}\right)$$

Parece que deberíamos de usar el caso 3, pero todavía debemos revisar que se cumpla la condición de regularidad para algún $c < 1$:

$$af\left(\frac{n}{b}\right) \leq cf(n) \Rightarrow$$

$$3\left(\frac{n}{4}\right)\log_2\left(\frac{n}{4}\right) = \frac{3}{4}n\log_2 n - \frac{3}{4}n\log_2 4 = \frac{3}{4}n\log_2 n - \frac{3}{2}n \leq cn\log_2 n$$

Podemos tomar cualquier c tal que $\frac{3}{4} \leq c < 1$, y la condición se cumple. Además, debemos recordar que se debe cumplir para n's suficientemente grandes, por lo que podríamos incluso determinar el n_0 que permita ajustar distintos valores de c. Al aplicar el caso 3, entonces, el resultado es:

$$T(n) = \Theta\big(f(n)\big) = \Theta(n \lg n)$$

Una relación de recurrencia para la que el *Master Method* no aplica es $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$ porque $f(n)$ no es polinomialmente más grande que $n^{\log_b a} = n$. Esto significa que aunque es más grande no lo es por un factor de n^{ϵ} para $\epsilon > 0$. Es decir:

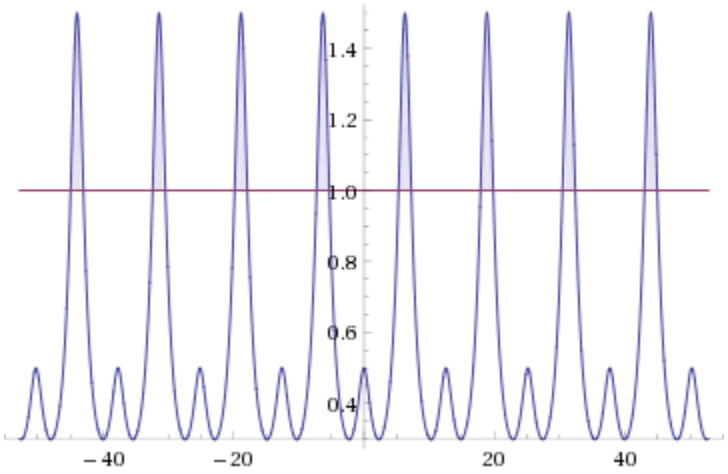
$$\frac{f(n)}{n^{\log_b a}} = \frac{n \lg n}{n} = \lg n \leq n^{\epsilon}$$

para cualquier $\epsilon > 0$ a partir de algún n_0. De igual forma la ecuación $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$ no cumple porque, aunque $n^{\log_b a} = 1$ y definitivamente $f(n) = n(2 - \cos n) = \Omega(n^{\log_b a + \epsilon})$ con $\epsilon = 1$, la condición de regularidad no cumple:

$$af\left(\frac{n}{b}\right) = 1 * \frac{n}{2}\left(2 - \cos\frac{n}{2}\right) = n - \frac{n}{2}\cos\frac{n}{2} = \frac{\left(2n - n\cos\frac{n}{2}\right)}{2} \leq cf(n) = cn(2 - \cos n) \Rightarrow$$

$$\frac{2 - \cos\frac{n}{2}}{2(2 - \cos n)} \leq c$$

Pero $\frac{2-\cos\frac{n}{2}}{2(2-\cos n)}$ es periódicamente mayor o igual a uno (ver gráfica a continuación) por lo que, sabiendo que $c < 1$, tenemos una contradicción.


1.4
1.2
1.0
0.8
0.6
0.4
−40
−20
20
40

Apéndice: deducción de la serie geométrica

Una serie geométrica $\sum_{i=0}^{n} ar^i$ se puede expresar como $a\left(\frac{r^{n+1}-1}{r-1}\right)$ siguiendo los siguientes pasos:

$$s = a + ar + ar^2 + \cdots + ar^n$$

$$rs = ar + ar^2 + ar^3 + \cdots + ar^{n+1}$$

$$rs - s = ar^{n+1} - a$$

$$s(r-1) = a(r^{n+1} - 1)$$

$$s = a\left(\frac{r^{n+1} - 1}{r - 1}\right)$$

Nótese que al multiplicar este resultado por $\frac{-1}{-1}$ se puede usar la forma $a\left(\frac{1-r^{n+1}}{1-r}\right)$, más conveniente para cuando $0 < r < 1$, que se alcanza mediante un procedimiento similar:

$$s = a + ar + ar^2 + \cdots + ar^n$$

$$rs = ar + ar^2 + ar^3 + \cdots + ar^{n+1}$$

$$s - rs = a - ar^{n+1}$$

$$s(1-r) = a(1 - r^{n+1})$$

$$s = a\left(\frac{1 - r^{n+1}}{1 - r}\right)$$

Fuentes:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms.* Massachusetts: The MIT Press.
- Soltys, M. (2012). *An Introduction to the Analysis of Algorithms.* Singapore: World Scientific Publishing.
- https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)