Chap2 Divide-and-Conquer

> **Info**
>
> 分治法，您可以遞歸地解決給定的問題（實例）。如果問題夠小，基本情況你直接解決它
> 無需遞迴。否則—在遞歸情況下—您將執行三個特徵步驟：
>
> 1. 將問題分為一個或多個子問題，這些子問題是同一問題的較小實例。
> 2. 透過遞歸解決子問題來解決它們。
> 3. 將子問題的解組合起來形成原問題的解。
>   分而治之演算法將一個大問題分解為更小的子問題，這些子問題本身可以分解為更小的子問題，
>   等等。當遞歸到達基本情況並且子問題足夠小可以直接求解而無需進一步遞歸時，遞歸就會觸底。

解決遞迴問題

我們需要簡單易用的工具來處理最常見的情況。但我們也希望通用工具能夠適用於較不常見的情況，也許需要付出更多的努力。本章提供了四種解決遞歸的方法：

1. 在替換法(substitution method)中，您猜測界限的形式，然後使用數學歸納法來證明您的猜測正確並求解常數。這種方法可能是解決遞歸問題的最可靠的方法，但它也需要您做出良好的猜測並產生歸納證明。
2. 遞歸樹(recursion-tree method)方法將遞歸建模為樹，其節點代表遞歸各個層級所產生的成本。為了解決遞歸問題，您可以確定每個等級的成本並將其相加，也許可以使用 A.2 節中的邊界求和技術。即使您不使用此方法來正式證明界限，它也有助於猜測替換方法中使用的界限的形式。
3. 當適用時，主方法是最簡單的方法。它提供了這個形式的界限:

$$T(n) = aT(b/n) + f(n)$$

   其中 a > 0 和 b > 1 是常數，f(n) 是給定的「驅動」函數。
   這種類型的重複在演算法研究中比其他任何類型的重複出現得更頻繁。它描述了一種分而治之演算法，該演算法創建一個子問題，每個子問題的大小都是原始問題大小的$1/b$，使用$f(n)$時間進行分割和組合步驟。要應用主方法，您需

要記住三種情況，但一旦記住，您就可以輕鬆確定許多分而治之演算法的運行時間漸近界限。

Recursion Tree的一些數字來由

“Pasted image 20240824094012.png” could not be found.

-$log_4 n$:每拆一次子問題大小就是上一層的$1/4$
-$3^{log_4 n}$:每拆一次，子問題都會翻3倍

- 註:$a^{logb} = b^{loga}$

合併成本:$\Theta(1) \cdot n^{log_4 3}$

老大定理(Master Theorem)

> **Info**
>
> 主遞歸描述了分而治之演算法的運行時間，該演算法將大小為 n 的問題分成子問題，每個子問題的大小為$n/b < n$。
> 此演算法遞歸地求解a個子問題，每個子問題需要$T(n/b)$時間。驅動函數$f(n)$包含在遞歸之前劃分問題的成本，以及將遞歸解決方案的結果組合到子問題的成本。例如，$Strassen$演算法產生的遞歸是 a = 7、b = 2 和驅動函數$f(n) = \Theta(n^2)$ 的主遞歸。正如我們所提到的，在求解描述演算法運行時間的遞歸式時，我們經常傾向於忽略的一個技術細節是"輸入大小n為整數"的要求。例如，我們看到歸併排序的運行時間可以用遞歸來描述:
>
> $$T(n) = 2T(n/2) + \Theta(n)$$
>
> 但如果n是奇數，我們確實不會遇到兩個大小剛好是一半的問題。相反，為了確保問題大小為整數，我們將一個子問題向下舍入到大小$\lfloor n/2 \rfloor$，將另一個子問題向上舍入到大小$\lceil n/2 \rceil$，因此真正的遞歸是
> $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$。但這種上下遞推比在實數上定義的遞推寫起來更長、處理起來更混亂。如果不需要的話，我們寧願不擔心取整，特別是因為兩個遞歸具有相同的$\Theta(n\ lgn)$解。
> 老大定理允許您聲明沒有下限和上限的主遞歸並隱式推斷它們。無論參數如何向上或向下舍入到最接近的整數，它提供的漸近界限保持不變。此外，正如我們將在第 4.6 節中看到的，如果你在實數上定義主遞推式，沒有隱含的下限和

上限，漸進界限仍然不會改變。因此，您可以忽略主重複的下限和上限。第 4.7 節給出了在更一般的分而治之遞歸中忽略下限和上限的充分條件。

用老大定理解遞迴

> ⓘ Info
>
> 令 a > 0 和 b > 1 為常數，並令$f(n)$為驅動函數，即在所有足夠大的實數上被定義且非負。
> 定義在$n \in \mathbb{N}$上的函數$T(n) = aT(n/b) + f(n)$

1. 若存在常數$\epsilon > 0$使得$f(n) = O(n^{\log_b a - \epsilon})$，則$T(n) = \Theta(n^{\log_b a})$。
2. 若存在常數$k \geq 0$使得$f(n) = \Theta(n^{\log_b a} \log^k n)$，則$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$。
3. 如果存在常數$\epsilon > 0$使得$f(n) = \Omega(n^{\log_b a + \epsilon})$，並且如果$f(n)$另外滿足$af(n/b) \leq cf(n)$對某個常數$c < 1$且所有夠大的$n$，則$T(n) = \Theta(f(n))$。

老大定理推導

在分治法 (Divide & Conquer) 中一個大問題可以分成數個小問題，接下來再結合這些小問題的答案 (Combine)，因此列出的遞迴式常像：
$T(n) = aT\left(\frac{n}{b}\right) + f(n)$
其中a代表分成了a個小問題，$\frac{n}{b}$是每個小問題的大小，$f(n)$是結合答案所需的時間。

經過下列運算，可以得到新的表示方式：

$$
\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + f(n);\ a, b \geq 1 \\
&= a\left[aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right] + f(n) \\
&= a^2T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) \\
&= \cdots (\text{let } n = b^i) \\
&= a^iT\left(\frac{n}{b^i}\right) + a^{i-1}f\left(\frac{n}{b^{i-1}}\right) + a^{i-2}f\left(\frac{n}{b^{i-2}}\right) + \cdots + f(n)
\end{aligned}
$$

每次展開後，首項中a的次方數就會增加一，不妨令$n = b^i$，代表這樣的展開總共可以進行幾次$(i次)$，最後可以得到上述的結果。

又因為：
$n = b^i \leftrightarrow i = \log_b n$
代入上式，得到：

$$
\begin{aligned}
T(n) = & \\
&= a^i T\left(\frac{n}{b^i}\right) + a^{i-1} f\left(\frac{n}{b^{i-1}}\right) + a^{i-2} f\left(\frac{n}{b^{i-2}}\right) + \cdots + f(n) \quad \text{---} \\
&= a^{\log_b n} T(1) + a^{i-1} f\left(\frac{n}{b^{i-1}}\right) + a^{i-2} f\left(\frac{n}{b^{i-2}}\right) + \cdots + f(n) \quad \text{---} \\
&= n^{\log_b a} T(1) + a^{i-1} f\left(\frac{n}{b^{i-1}}\right) + a^{i-2} f\left(\frac{n}{b^{i-2}}\right) + \cdots + f(n) \quad \text{---}
\end{aligned}
$$

註：$a^{logb} = b^{loga}$

但是要把所有項加起來需要另外$\Theta(n^{\log_b a})$ 合併成本，因此:

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

但其實就算把 $\Theta(n^{log_b a})$ 考量進去,你會發現這項 $\Theta(n^{log_b a})$ 其實不影響到最後運算的結果 。
為以 $Big—O$ 求時間複雜度時，只需考慮n的指數最大的那項，因此觀察展開後的式子，我們可以只關注(c)式中第一項$n^{\log_b a} T(1)$ 和最後一項$f(n)$之間誰大誰小即可

經典例題

1. The Maximum Subarray Problem
2. Matrix Multiplication