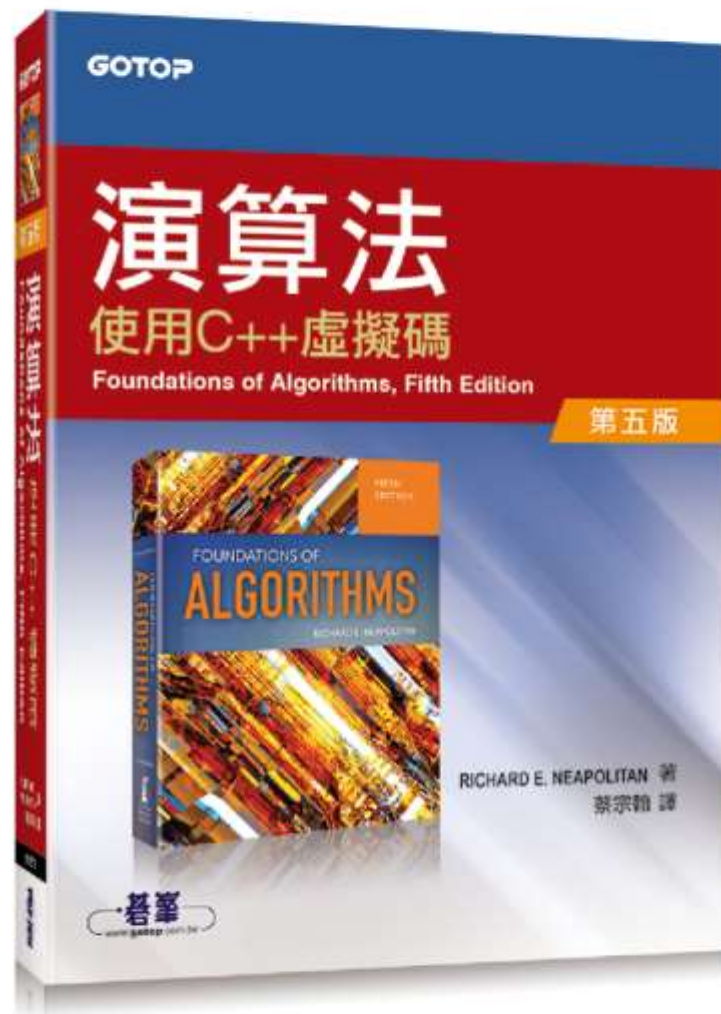


第二章

Divide-and-Conquer



第二章 Divide-and-Conquer

- [2.1 二元搜尋法](#)
- [2.2 合併搜尋法](#)
- [2.3 Divide-and-conquer技巧](#)
- [2.4 快速排序法\(分割交換排序法\)](#)
- [2.5 Strassen的矩陣相乘演算法](#)
- [2.6 大整數的計算](#)
 - [2.6.1大整數的表達法：加法與其他線性時間的運算](#)
 - [2.6.2大整數的乘法](#)
- [2.7決定門檻值](#)
- [2.8何時不能使用Divide-and-Conquer](#)

Divide-and-Conquer (各個擊破)

- 將一個問題切成兩個或以上的較小的問題。較小的問題通常是原問題的實例。
- 如果較小的問題的解可以容易地獲得，那麼原問題的解可以藉由合併小問題的答案獲得。
- 如果小問題還是太大以致於不易解決，則可以再被切成更小的問題。經由該種切割的過程直到切到夠小能夠求解為止

2.1 二元搜尋法

如果 x 與中間項相同，離開。否則：

1. 將該陣列分割(*Divide*)成約一半大小的兩個子陣列。
 - 如果 x 小於中間項，選擇左邊的子陣列。
 - 如果 x 大於中間項，則選擇右邊的子陣列。
2. 藉由判斷 x 是否在該子陣列中來克服(*Conquer*)該子陣列。除非該子陣列夠小，否則使用遞迴來做這件事。
3. 由子陣列的解答獲得(*Obtain*)該陣列的解答。

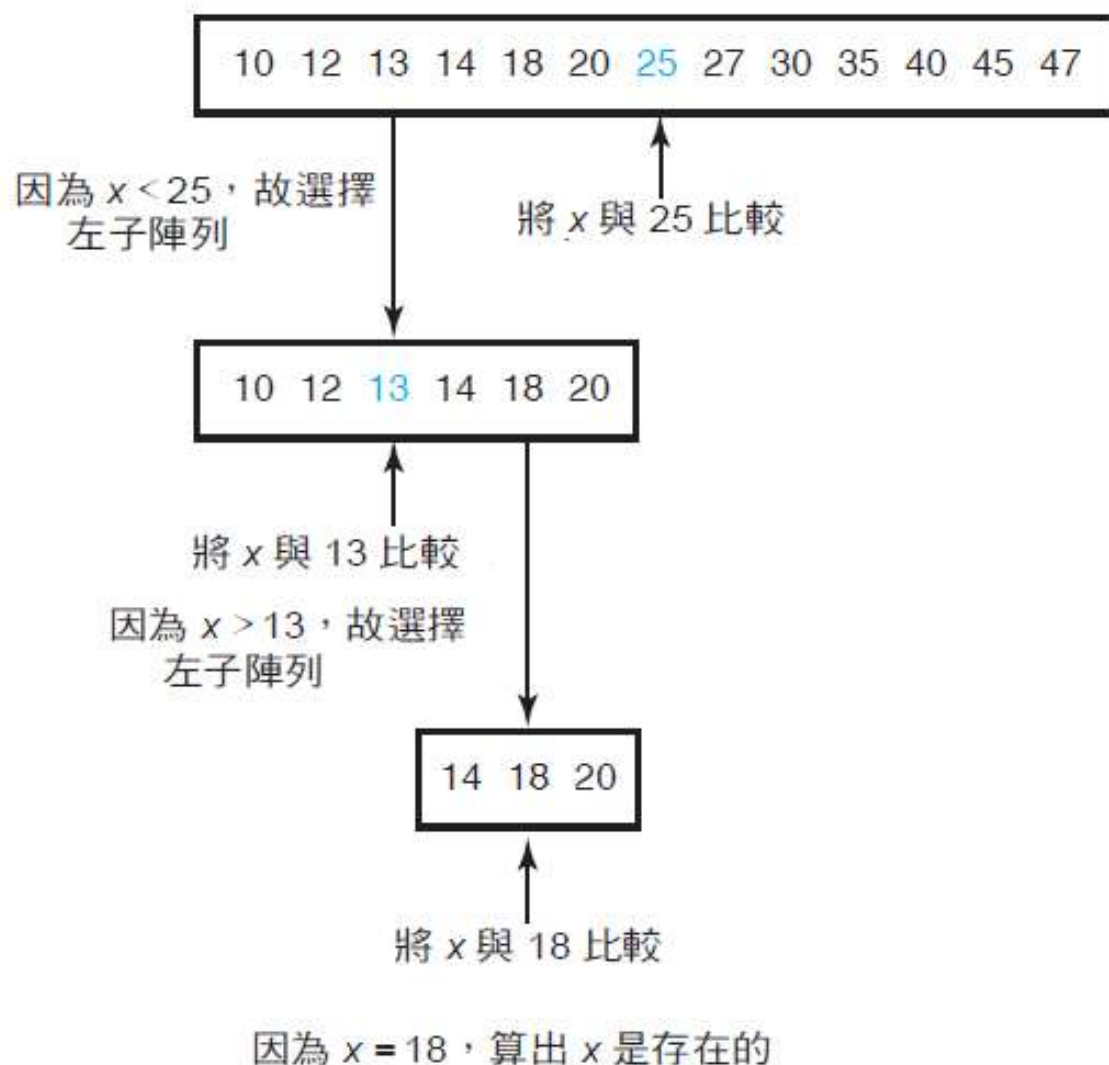


圖 2.1 使用二元搜尋法搜尋時，執行者所操作的步驟。(注意： $x=18$ 。)

演算法 2.1 二元搜尋法(遞迴版)

問題：判斷 x 是否在大小為 n 的已排序陣列 S 中。

輸入：正整數 n ，以非遞減的順序排序的陣列 S (索引值由 1 到 n)，以及 $\text{key } x$ 。

輸出： $\text{location } x$ 在 S 中的位置 (如果 x 不在 S 中，將傳回 0)。

```
index location (index low, index high)
{
    index mid;

    if (low > high)
        return 0;
    else {
        mid =  $\lfloor (low + high) / 2 \rfloor$ ;
        if ( $x == S[mid]$ )
            return mid
        else if ( $x < S[mid]$ )
            return location (low, mid - 1);
        else
            return location (mid + 1, high);
    }
}
```

Recursion vs. Iterative

- 此二元搜尋法的遞迴版本採用tail-recursion
- 利用iteration來取代tail-recursion是相當有利的
- 因為iterative演算法並不需要去維護堆疊(stack)，因此其執行速度較遞迴版本快速

分析演算法 2.1

最差情況的時間複雜度 (二元搜尋法，遞迴版)

- **基本運算**： x 與 $S[mid]$ 的比較。
- **輸入大小**： n ，陣列中的項目數量。
- 如同在 1.2 節中討論的，一個最壞的情況是 x 較所有陣列中的項目為大。若 n 是 2 的乘幂且 x 比所有陣列中的項目都大，每次遞迴呼叫可以減少的個數正好一半。例如，若 $n=16$ ，則 $mid = \lfloor (1+16)/2 \rfloor = 8$ 。因為 x 大於所有陣列中的項目，前 8 大的項目成為第一次的遞迴呼叫的輸入。同樣地，前四大的項目是第二次遞迴呼叫的輸入。我們得到下列的遞迴式：

$$W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{在遞迴呼叫中的比較}} + \underbrace{1}_{\text{在最上層的比較}}$$

- 如果 $n=1$ 且 x 大於一個陣列項目，會造成 x 與該項目比較之後，將跟著一個 $low > high$ 的遞迴呼叫。此時，終止條件為真，意味著比較到此為止。因此， $W(1)$ 等於 1。我們得到下列的遞迴式

$$\begin{aligned} W(n) &= W\left(\frac{n}{2}\right) + 1 \quad \text{對於 } n > 1, n \text{ 為 } 2 \text{ 的乘幂} \\ W(1) &= 1 \end{aligned}$$

- 我們在附錄 B 中的範例 B.1 將會解出這個遞迴式。其解答為

$$W(n) = \lg n + 1$$

- 如果 n 不限為 2 的乘幂，我們可得到

$$W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$$

2.2 合併排序法(merge sort)

1. 將該陣列分割(*Divide*)成為兩個具有 $n/2$ 個項目的子陣列。
2. 藉由排序以解答(*Conquer*)每個子陣列，除非該陣列夠小，否則使用遞迴來完成這個動作。
3. 將子陣列合併成一個已排序的陣列以合併(*Combine*)子陣列的解答

範例 2.2

假設一個陣列包含了下面的數字，依序為：

27 10 12 20 25 13 15 22

1. 分割 (Divide) 這個陣列：

27 10 12 20 及 25 13 15 22

2. 對每個子陣列進行排序：

10 12 20 27 及 13 15 22 25

3. 合併子陣列：

12 13 15 20 22 25 27

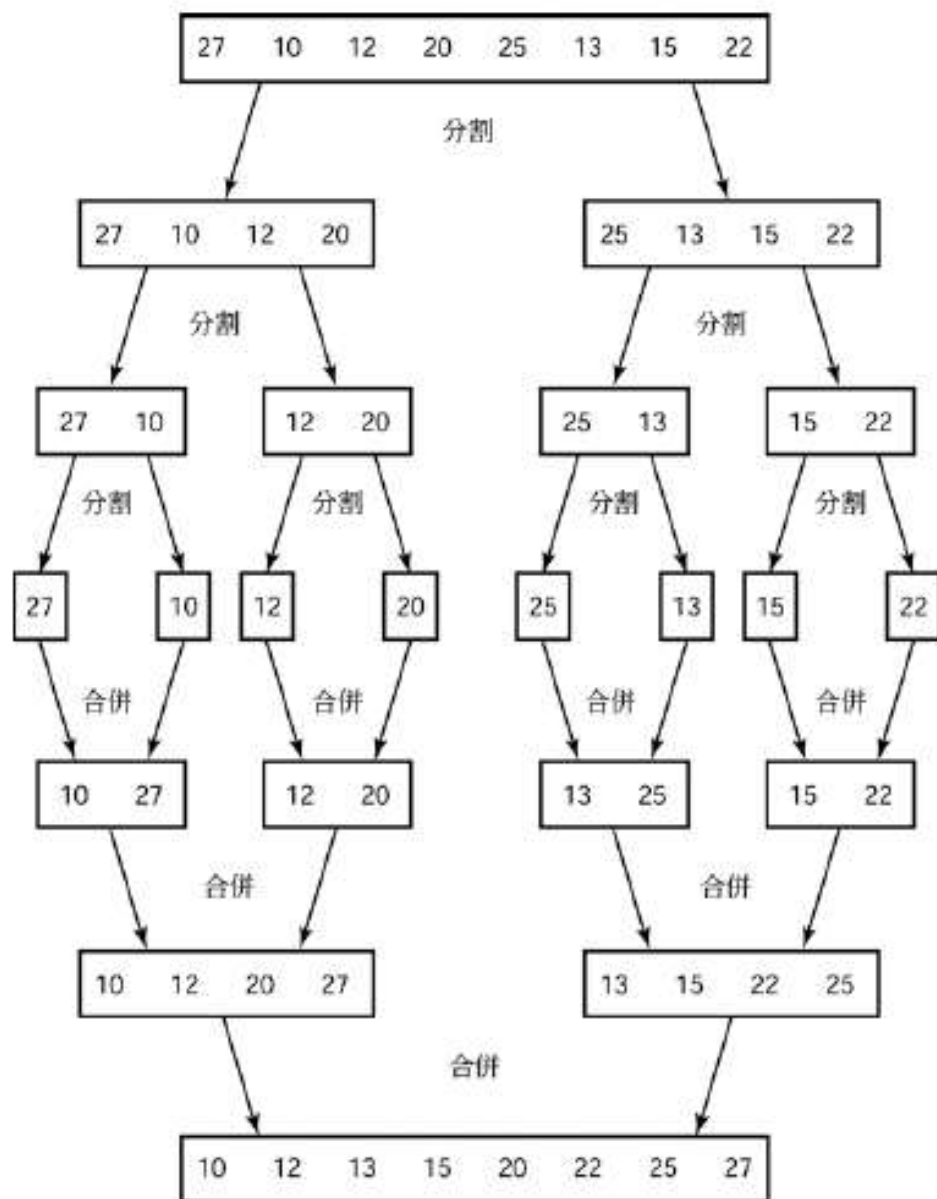


圖 2.2 • 使用合併排序時，執行者所操作的步驟。

演算法 2.2 合併排序

問題：將 n 個 key 排序成為非遞減序列。

輸入：正整數 n ，鍵值陣列 S (索引值由 1 到 n)。

輸出：陣列 S ，其包含的 key 已經以非遞減的順序排序過。

```
void mergesort (int n, keytype S[])
{
    if (n>1) {
        const int h =  $\lfloor n/2 \rfloor$ , m = n - h;
        keytype U[1..h], V[1..m];
        將 S[1] 至 S[h] 複製到 U[1] 至 U[h] 間;
        將 S[h+1] 至 S [ $n$ ] 複製到 V[1] 至 V[m] 間;
        mergesort (h, U);
        mergesort (m, V);
        merge (h, m, U, V, S);
    }
}
```

演算法 2.3 合併

問題：將兩個已排序的陣列合併成一個已排序的陣列。

輸入：正整數 h 與 m ，已排序的陣列 U (索引值由 1 到 h)，已排序的陣列 V (索引值由 1 到 m)。

輸入：單一已排序的陣列 S (索引值由 1 到 $h+m$)，其包含了 U 與 V 的所有 key。

```
void merge (int h, int m, const keytype U[],  
            const keytype V[],  
            keytype S[])
```



```
{
    index i, j, k;

    i = 1; j = 1; k = 1;
    while (i <= h && j <= m) {
        if (U[i] < V[j]) {
            S[k] = U[i];
            i++;
        }

        else {
            S[k] = V[j];
            j++;
        }
        k++;
    }
    if (i > h)
        將 V[j] 至 V[m] 複製到 S[k] 至 S[h+m] 間 ;
    else
        將 U[i] 至 U[h] 複製到 S[k] 至 S[h+m] 間 ;
}
```

分析演算法 2.3

最差情況的時間複雜度 (合併)

基本運算：比較 $U[i]$ 與 $V[j]$ 。

輸入大小： h 與 m ，分別為兩輸入陣列的項目個數。

最壞的狀況發生在當跳出迴圈時，因為 i 已經到達離開點 $h+1$ ，而另一個索引 j 只到達 m ，還差 1 才到達離開點。舉例來說，若在 S 中先放 $m-1$ 個 V 中的項目，再放 U 中的所有 h 個項目，由於 i 等於 $h+1$ ，此時將會跳出迴圈。因此，

$$W(h, m) = h + m - 1$$

分析演算法 2.2

最差情況的時間複雜度 (合併排序)

基本運算：發生在 *merge* 中的比較。

輸入大小： n ，陣列 S 中的項目個數。

所有比較的數目為以 U 為輸入對 *mergesort* 的遞迴呼叫，以 V 為輸入對 *mergesort* 的遞迴呼叫以及最上層對 *merge* 的呼叫，三者中所發生的比較次數總和。因此，

$$W(n) = \underbrace{W(h)}_{\text{將 } U \text{ 排序所花}} + \underbrace{W(m)}_{\text{將 } V \text{ 排序所花}} + \underbrace{h + m - 1}_{\text{合併所花}}$$

的時間 的時間 的時間

- n 為2的乘冪的情況

$$h = \lfloor n/2 \rfloor = \frac{n}{2}$$

$$m = n - h = n - \frac{n}{2} = \frac{n}{2}$$

$$h + m = \frac{n}{2} + \frac{n}{2} = n$$

- $W(n)$ 的式子成為

$$\begin{aligned} W(n) &= W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1 \\ &= 2W\left(\frac{n}{2}\right) + n - 1 \end{aligned}$$

當輸入的大小為 1 時，即達到終止條件，且無法再進行合併。因此， $W(1)$ 為 0。我們得到下列的遞迴式

$$\begin{aligned} W(n) &= 2W\left(\frac{n}{2}\right) + n - 1 \quad \text{當 } n > 1 \text{ 且 } n \text{ 為 } 2 \text{ 的乘幂} \\ W(1) &= 0 \end{aligned}$$

這個遞迴式將在附錄 B 的範例 B.19 解出。其解答為

$$W(n) = n \lg n - (n - 1) \in \Theta(n \lg n)$$

- 原地置換排序(in-place sort)
 - 不需要用到存放輸入資料之額外空間
 - 演算法2.2並不是，因為除了輸入陣列 S 外，它使用兩個額外陣列 U 與 V
 - 建立額外的陣列項目的總和為 $n(1 + 1/2 + 1/4 + \dots) = 2n$
 - 將額外的空間減低到 n 個項目是有可能的。我們必須對輸入陣列 S 做更多的處理才能達到這項要求。(如演算法2.4)

演算法 2.4 合併排序2

問題：將 n 個 key 排序成為非遞減序列。

輸入：正整數 n ，key 陣列 S (索引值由 1 到 n)。

輸出：key 陣列 S ，其包含所有的 key 已經以非遞減的順序排序過。

```
void mergesort2 (index low, index high)
{
    index mid;

    if (low < high) {
        mid =  $\lfloor (low + high) / 2 \rfloor$ ;
        mergesort2 (low, mid);
        mergesort2 (mid + 1, high);
        merge2 (low, mid, high);
    }
}
```

演算法 2.5 合併2

問題：合併兩個在合併排序 2 中產生， S 的已排序子陣列。

輸入：索引值 low 、 mid 與 $high$ ， S 的子陣列（索引值由 low 到 $high$ ）。其中在 low 到 mid 與 $mid+1$ 到 $high$ 這兩個區間的 key 都已經以非遞減的順序排好。

輸出： S 的子陣列（索引值由 low 到 $high$ ），其中在 low 到 $high$ 區間的 key 已經以非遞減的順序排好。

```
void merge2 (index low, index mid, index high)
{
    index i, j, k;
    keytype U[low..high];           // 合併所需用到的區域陣列變數

    i = low; j = mid + 1; k = low;
    while (i ≤ mid && j ≤ high) {
        if (S[i] < S[j]) {
            U[k] = S[i];
            i++;
        }
        else {
            U[k] = S[j];
            j++;
        }
        k++;
    }
    if (i > mid)
        將 S[j] 至 S[high] 複製到 U[k] 至 U[high] 間 ;
    else
        將 S[i] 至 S[mid] 複製到 U[k] 至 U[high] 間 ;
    將 U[low] 至 U[high] 複製到 S[low] 至 S[high] 間 ;
}
```

2.3 Divide-and-Conquer技巧

- 分割(*Divide*)一個較大問題實例成為一個或多個較小的實例。
- 解出每個較小實例的答案(*Conquer*)。
 - 除非實例已經分割到足夠小的地步，否則使用遞迴來解。
- 必要的話，將兩個較小實體的解合併(*Combine*)以獲得原始問題實例的解答。
 - 在某些演算法，例如Binary Search Recursive(演算法2.1)實體只有被縮減成一個較小的實體，所以不需要去合併解答。

2.4 快速排序法 (分割交換排序法)

- 陣列會將小於某個樞紐(pivot)的項目都放在前面，而把大於樞紐的項目都放在後面
- 樞紐可以為任意的項目，為簡單起見，通常可挑選第一個項目做為樞紐
- Ref.
<http://notepad.yehyeh.net/Content/Algorithm/Sort/Quick/Quick.php>

範例 2.3

假設某個陣列包含了下列的數值：

樞紐項
↓
15 22 13 27 12 10 20 25

1. 分割該陣列，將小於樞紐的項目移至它的左邊而大於樞紐的項目則移至它的右邊

樞紐項
↓
全部 10 13 12 15 22 27 20 25 的項
樞紐項

2. 將子陣列排序：

↓
10 12 13 15 20 22 25 27
已排序 已排序

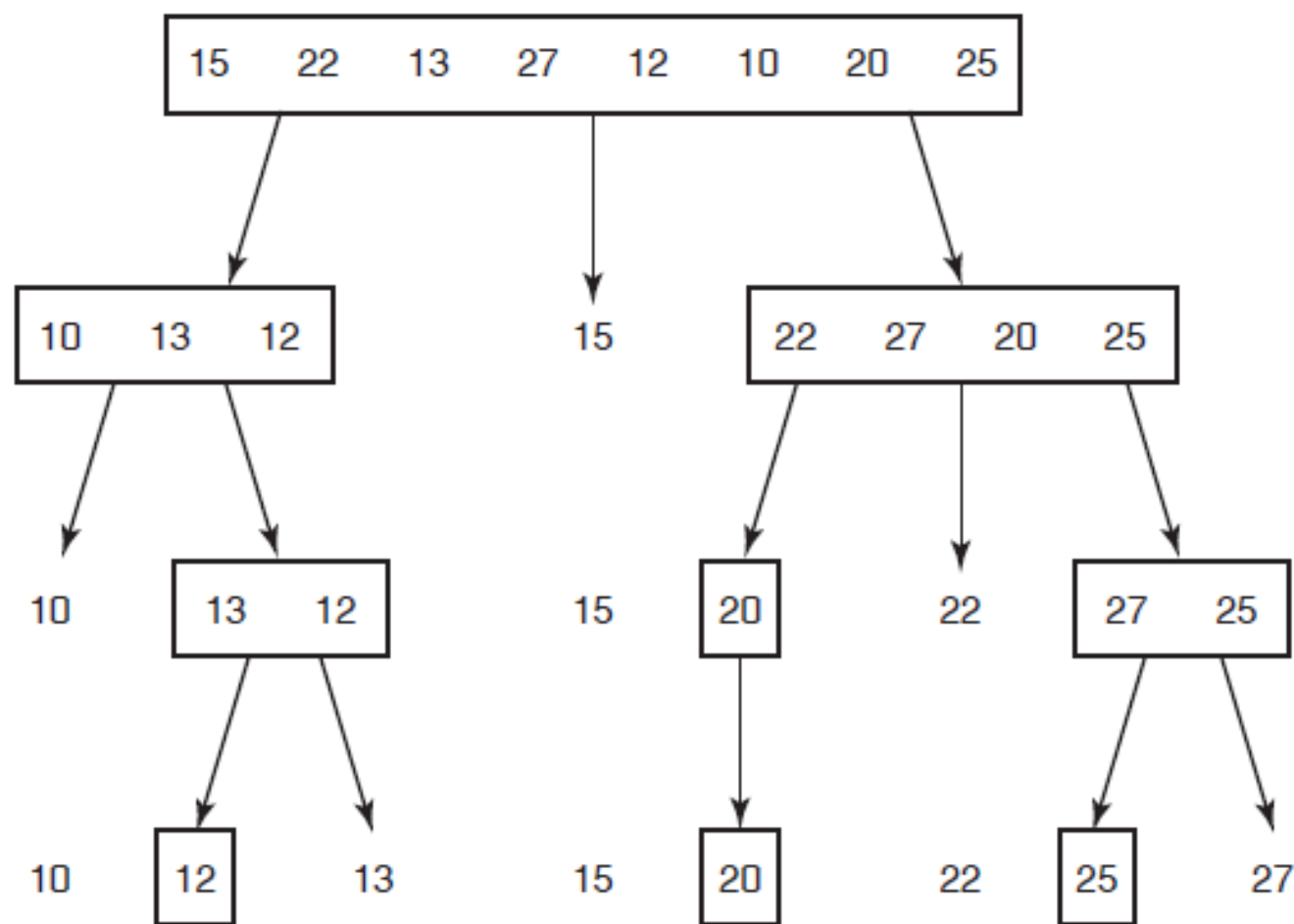


圖 2.3 人操作快速排序法執行排序的步驟。子陣列被用方框包起來，而樞紐項則沒有。

演算法 2.6 快速排序

問題：將 n 個 key 排序成為非遞減序列。

輸入：正整數 n ，key 陣列 S (索引值由 1 到 n)。

輸出：key 陣列 S ，其包含所有的 key 已經以非遞減的順序排序過。

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low) {
        partition (low, high, pivotpoint);
        quicksort (low, pivotpoint - 1);
        quicksort (pivotpoint + 1, high);
    }
}
```

演算法 2.7 分割

問題：將 n 個 key 排序成為非遞減序列。

輸入：正整數 n ，key 陣列 S (索引值由 1 到 n)。

輸出：key 陣列 S ，其包含所有的 key 已經以非遞減的順序排序過。

```
void partition (index low, index high,
                index& pivotpoint)
{
    index i, j;
    keytype pivotitem;

    pivotitem = S[low];           // 選擇第一個項目做為 pivotitem (樞紐項)
    j = low;
    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem) {
            j++;
            交換 S[i] 與 S[j] 的內容;
        }
    pivotpoint = j;
    交換 S[low] 與 S[pivotpoint] 的內容;    // 將 pivotitem 放在 pivotpoint
}
```

分析演算法 2.7

所有情況的時間複雜度 (分割)

基本運算：發生在 *merge* 中的比較。

輸入大小： $n = high - low + 1$ ，也就是在子陣列中的項目數。

因為除第一個項目外，每個項目都會被比較，

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

因此我們使用 n 來代表子陣列的大小而非陣列 S 的大小。只有當 *partition* 在最上層被呼叫時，它才代表 S 的大小。

分析演算法2.6

最差情況的時間複雜度 (快速排序)

基本運算：在 *partition* 副程式中， $S[i]$ 與樞紐項目的比較。

輸入大小： n ，陣列 S 中的項目數。

- 最差情況：該陣列已經排成非遞減的順序

陣列不斷地被切成兩個子陣列。左邊的陣列是空的，而右邊的陣列則比原來少一個項目。

$$T(n) = \underbrace{T(0)}_{\text{排列左邊子陣列的時間}} + \underbrace{T(n-1)}_{\text{排列右邊子陣列的時間}} + \underbrace{n-1}_{\text{分割所需的時間}}$$

- 因為 $T(0)=0$ ，故可以得到下列的遞迴式

$$\begin{aligned} T(n) &= T(n-1) + n - 1 & \text{for } n > 0 \\ T(0) &= 0 \end{aligned}$$

- 根據附錄B的範例B.18

$$T(n) = \frac{n(n-1)}{2}$$

2.5 Strassen的矩陣相乘演算法

- 演算法1.4(矩陣乘法)
 - 完全依照矩陣乘法的定義求得兩個矩陣相乘的結果。我們已證明其使用的乘法次數的時間複雜度為 $T(n) = n^3$ ，其中 n 為矩陣的列數與欄
 - 這個演算法很快地就失去了實用價值
- 在1969年，Strassen發表了一個演算法。無論是以乘法次數或是加減法次數來評估，該演算法的時間複雜度均較前述的三次方演算法為佳

範例 2.4

假設我們想得到 A 與 B 兩個 2×2 的矩陣的乘積 C ，也就是說，

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Strassen 證明假設我們令

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

$$m_2 = (a_{21} + a_{22})b_{11}$$

$$m_3 = a_{11}(b_{12} - b_{22})$$

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

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

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

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

則乘積 C 可由下列式子求得

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

$$\begin{array}{c} \updownarrow n/2 \\ \leftarrow n/2 \rightarrow \end{array} \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

圖 2.4 在 Strassen 演算法中將大矩陣分割成子矩陣的動作。

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1,n/2} \\ a_{21} & a_{22} \cdots a_{2,n/2} \\ \vdots & \vdots \\ a_{n/2,1} & \cdots a_{n/2,n/2} \end{bmatrix}$$

- 利用 Strassen 的方法，首先我們計算

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

- 在此我們的運算變成了矩陣的加法與乘法。以同樣的方式，我們可以算出到。接下來我們計算

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

- 與 C_{12} 、 C_{21} 與 C_{22} 。最後，A與B的乘積C可由合併四個子矩陣得到。下面的例子描述了這些步驟。

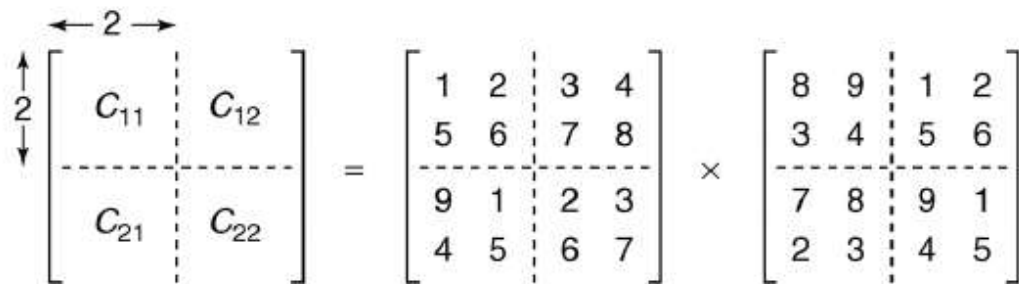
範例 2.5

假設

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

圖2.5描述了Strassen方法中的分割動作。其計算進行如下：

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ &= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right) \\ &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \end{aligned}$$



$$\begin{array}{c} \leftarrow 2 \rightarrow \\ \uparrow 2 \downarrow \end{array} \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \times \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

圖 2.5 給定 $n=4$ 及矩陣的內容，在 Strassen 演算法中將大矩陣分割成子矩陣的動作

當矩陣已經足夠小了，我們就用標準的方法來相乘。在本例中，我們是在 $n=2$ 時開始使用標準相乘法。於是，

$$\begin{aligned} M_1 &= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} \\ &= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} \end{aligned}$$

在此之後，我們以同樣的方法計算 M_2 到 M_7 ，接著 C_{11} 、 C_{12} 、 C_{21} 與 C_{22} 的值都會計算出來。將它們合起來可以得到 C 。

演算法 2.8 Strassen

問題：當 n 為 2 的乘幂時，求出兩個 $n \times n$ 矩陣的乘積。

輸入：一個為 2 的乘幂的整數 n ，以及兩個 $n \times n$ 矩陣 A 與 B 。

輸出： A 與 B 的乘積 C 。

```
void strassen (int n
                n × n_matrix A,
                n × n_matrix B,
                n × n_matrix& C)
{
    if (n <= threshold)
        使用標準演算法計算  $C = A \times B$ ;
    else {
        將  $A$  切成 4 個子矩陣  $A_{11}, A_{12}, A_{21}, A_{22}$ ;
        將  $B$  切成 4 個子矩陣  $B_{11}, B_{12}, B_{21}, B_{22}$ ;
        使用 Strassen 的方式計算  $C = A \times B$ ;
        // 遞迴呼叫範例 ;
        // strassen (n/2,  $A_{11} + A_{22}, B_{11} + B_{22}, M_1$ );
    }
}
```


分析演算法2.8

乘法次數的所有情況的時間複雜度分析 (Strassen)

基本運算：一個基本的乘法。

輸入大小： n ，也就是這些矩陣的列數與欄數。

為簡化起見，在分析這個情況時，我們會一直將矩陣分割下去，直到得到兩個 1×1 矩陣為止；到此，我們只需要把兩個矩陣裡面的數字相乘。實際上使用的門檻值並不會影響量級。當 $n=1$ 時，只需要做一次的乘法。當輸入是兩個 $n \times n$ 的矩陣時 ($n > 1$)，這個演算法會被呼叫 7 次，每次都會傳遞一個 $(n/2) \times (n/2)$ 的矩陣，並且在最上層時並不會用到乘法。故可得下列的遞迴式

$$\begin{aligned} T(n) &= 7T\left(\frac{n}{2}\right) \quad \text{當 } n > 1 \text{ 且 } n \text{ 為 } 2 \text{ 的乘幂} \\ T(1) &= 1 \end{aligned}$$

在附錄 B 的範例 B.2 解出這個遞迴式。答案如下

$$T(n) = n^{\lg 7} \approx n^{2.81} \in \Theta(n^{2.81})$$

2.6 大整數的計算

2.6.1 大整數的表達法：加法與其他線性時間的運算

- 例如，整數543,127可以表示為下面的陣列S：

5	4	3	1	2	7
<u>S[6]</u>	<u>S[5]</u>	<u>S[4]</u>	<u>S[3]</u>	<u>S[2]</u>	<u>S[1]</u>

- 想要具有表達正負整數的能力，我們只要保留在高位的陣列單元給正負號即可

2.6.2 大整數的乘法

我們將要發展一種比平方時間更快的演算法。我們的演算法是根據divide-and-conquer將一個 n 位數分割成兩個約為 $n/2$ 位數的整數

$$\underbrace{567,832}_{6 \text{ 位數}} = \underbrace{567}_{3 \text{ 位數}} \times 10^3 + \underbrace{832}_{3 \text{ 位數}}$$

$$\underbrace{9,423,723}_{7 \text{ 位數}} = \underbrace{9423}_{4 \text{ 位數}} \times 10^3 + \underbrace{723}_{3 \text{ 位數}}$$

如果 n 為整數 u 的位數，我們將把該整數分成兩個整數，一個是位數，一個是位數，如下所示：

$$\underbrace{u}_{n \text{ 位數}} = \underbrace{x}_{\lfloor n/2 \rfloor \text{ 位數}} \times 10^m + \underbrace{y}_{\lfloor n/2 \rfloor \text{ 位數}}$$

指數 m 可由下式求得

$$m = \left\lfloor \frac{n}{2} \right\rfloor$$

- 假定有兩個 n 位數整數如下

$$u = x \times 10^m + y$$

$$v = w \times 10^m + z$$

- 它們的乘積則為

$$\begin{aligned} uv &= (x \times 10^m + y)(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz \end{aligned}$$

- 在位數約為原來一半的整數上做4次相乘運算以及一些線性時間的運算，就可以得到 u 與 v 相乘的結果

範例 2.6

考慮下面的式子：

$$\begin{aligned} 567,832 \times 9,423,723 &= (567 \times 10^3 + 832)(9423 \times 10^3 + 723) \\ &= 567 \times 9423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723 \end{aligned}$$

遞迴做下去，這些較小整數的乘積可藉由將它們切成更小整數而得到。這個分割的程序將一直持續下去直到到達門檻值，然後再用標準的乘法計算方法。

演算法2.9 大整數乘法

問題：將兩個大整數 u 與 v 相乘。

輸入：大整數 u 與 v 。

輸出： u 與 v 的乘積 $prod$ 。

```
large_integer prod (large_integer u, large_integer v)
{
    large_integer x, y, w, z;
    int n, m;

    n = maximum(u的位數, v的位數)
    if (u == 0 || v == 0)
        return 0;
    else if (n <= threshold)
        return 由一般方法算出之  $u \times v$  的值;
```

```
else {  
    m = ⌊n/2⌋;  
    x = u divide 10m; y = u rem 10m;  
    w = v divide 10m; z = v rem 10m;  
    return prod (x, w) × 102m + (prod (w, y)) × 10m + prod (y, z);  
}  
}
```

prod(x, z)+

$$\begin{aligned} uv &= (x \times 10^m + y)(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz \end{aligned}$$

分析演算法 2.9

最差情況的時間複雜度 (大整數的乘法)

基本運算：當相加、相減、或執行 `divide` 10^m 、`rem` 10^m 、或 $\times 10^m$ 時，一位數字 (以十進位表示) 的操作。每次對後三者函式的呼叫導致基本運算執行 m 次。

輸入大小：也就是兩個大整數 u 、 v 的位數。

由於遞迴只有過了 *threshold* 才會停止，故最差的情況則發生在兩個整數都沒有出現 0 的位數時。我們將分析這個情況。

假設 n 為 2 的乘幂，那麼 x 、 y 、 w 與 z 的位數恰好均為 $n/2$ ，這意味著對 *prod* 的四個遞迴呼叫的輸入大小都是 $n/2$ 。因為 $m=n/2$ ，所以加法、減法、`divide` 10^m 、`rem` 10^m 、或 $\times 10^m$ 等線性時間的運算都具有以 n 為單位的線性時間複雜度。對這些線性時間運算來說，它們最大輸入大小並不相同，因此無法直接求得精確的時間複雜度。更簡單的方法，是將所有的線性時間運算組成一項 cn ，其中 c 為正的常數。於是我們的遞迴式成為

$$\begin{aligned} W(n) &= 4W\left(\frac{n}{2}\right) + cn && \text{當 } n > s \text{ 且 } n \text{ 為 } 2 \text{ 的乘幂} \\ W(s) &= 0 \end{aligned}$$

- 因為我們討論的是所有輸入的大小均為2的乘冪的情況，故 s 的實際值，就是當我們停止分割實體時那個 n 值。它的小於或等於 $threshold$ ，並為2的乘冪。
- 當 n 不限於2的乘冪的情況，建立的式子仍與前面的遞迴式類似，但式子內會包含下限整數(floor)與上限整數(ceiling)。利用類似於附錄B的範例B.25的歸納法證明，可以得知 $W(n)$ 是非遞減函數。於是，附錄B中的定理B.6意味著

$$W(n) \in \Theta(n^{\lg^4}) = \Theta(n^2)$$

2.7 決定門檻值

- n 的最佳門檻值(optimal threshold value)。
 - 當實例大小較該值小時，呼叫其他的替代演算法至少和繼續分割下去執行divide-and-conquer演算法一樣快
 - 當實例大小大於這個值時，繼續執行divide-and-conquer演算法
- 利用合併排序法與交換排序法來說明
 - 最差情況的時間複雜度

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

實作合併排序2(演算法2.4)

- 在實作的電腦上，合併排序2花在分割與重新合併大小為 n 的實例的時間為 $32n \mu s$
- 在這台電腦上，這個實作法的最差情況時間複雜度為：

$$W(n) = W(\lfloor \frac{n}{2} \rfloor) + W(\lceil \frac{n}{2} \rceil) + 32n \mu s$$

- 因為在輸入大小為1時，唯一所做的事只有終止條件的檢查因此我們假設 $W(1)$ 為0。簡化起見，只討論 n 為2的乘幂的情況。我們可以得到下列的遞迴式：

$$W(n) = 2W(n/2) + 32n \mu s \quad \text{當 } n > 1 \text{ 且 } n \text{ 為 } 2 \text{ 的乘幂時}$$

$$W(1) = 0 \mu s$$

- 根據附錄B

$$W(n) = 32n \lg n \mu s$$

為演算法2.5(合併排序2)決定呼叫演算法1.3(交換排序)的最佳門檻值

- 修改合併排序2，使得修正版的演算法會在 (t 為某個門檻值) 呼叫交換排序。假設我們是在剛剛討論的電腦上實作，對這個合併排序2的修訂版來說：

$$W(n) = \begin{cases} \frac{n(n-1)}{2} \mu s & \text{對於 } n \leq t \\ W(\lfloor \frac{n}{2} \rfloor) + W(\lceil \frac{n}{2} \rceil) + 32n \mu s & \text{對於 } n > t \end{cases} \quad (2.5)$$

- 欲求得 t 的最佳值，我們必須解出下列的式子

$$W(\lfloor \frac{t}{2} \rfloor) + W(\lceil \frac{t}{2} \rceil) + 32t = \frac{t(t-1)}{2} \quad (2.6)$$

- 因為 $\lfloor t/2 \rfloor$ 與 $\lceil t/2 \rceil$ 都小於等於 t ，因此不管實例的大小為兩者中的哪一個，都可以用不等式2.5中上面的不等式算出執行時間。因此

$$W\left(\lfloor \frac{t}{2} \rfloor\right) = \frac{\lfloor t/2 \rfloor (\lfloor t/2 \rfloor - 1)}{2} \quad \text{以及} \quad W\left(\lceil \frac{t}{2} \rceil\right) = \frac{\lceil t/2 \rceil (\lceil t/2 \rceil - 1)}{2}$$

代入方程式2.6可得

$$\frac{\lfloor t/2 \rfloor (\lfloor t/2 \rfloor - 1)}{2} + \frac{\lceil t/2 \rceil (\lceil t/2 \rceil - 1)}{2} + 32t = \frac{t(t-1)}{2} \quad (2.7)$$

- 將 t 以偶數代入方程式2.7， $\lfloor t/2 \rfloor$ 與 $\lceil t/2 \rceil$ 都等於 $t/2$ 。

$$t = 128$$

- 將 t 以奇數代入方程式2.7， $\lfloor t/2 \rfloor = (t-1)/2$ ， $\lceil t/2 \rceil = (t+1)/2$ 。

$$t = 128.008$$

於是，我們可獲得到了最佳門檻值128

範例 2.8

假設一個給定的 **divide-and-conquer** 演算法在某台特殊的電腦上執行的時間為

$$T(n) = 3T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 16n \mu s$$

其中 $16n \mu s$ 是分割與重新合併一個大小為 n 的實例所需花費的時間。假設在同樣的電腦上，某個 **iterative** 的演算法花了 $n^2 \mu s$ 來處理一個大小為 n 的實例。欲求出我們應該呼叫這個 **iterative** 演算法的 t 值，我們必須解出

$$3T\left(\left\lceil \frac{t}{2} \right\rceil\right) + 16t = t^2$$

因 $\lceil t/2 \rceil \leq t$ ，故此時會呼叫 **iterative** 演算法，這代表

$$T\left(\left\lceil \frac{t}{2} \right\rceil\right) = \left\lceil \frac{t}{2} \right\rceil^2$$

接著，求出下列方程式的解

$$3 \left\lceil \frac{t}{2} \right\rceil^2 + 16t = t^2$$

若將 t 以偶數代入（令 $\lceil t/2 \rceil$ 等於 $t/2$ ）並且解方程式可得

$$t = 64$$

若將 t 以奇數代入（令 $\lceil t/2 \rceil$ 等於 $(t+1)/2$ ）並解方程式可得

$$t = 70.04$$

因為這兩個 t 值不相等，因此最佳門檻值是不存在的。這代表著如果輸入大小為 64 到 70 間的偶數，再分割實例一次會較有效率；當輸入大小為 64 到 70 間的奇數，呼叫 **iterative** 演算法會較有效率。當輸入大小小於 64，呼叫 **iterative** 演算法一定效率較高；而當輸入大小大於 70，再分割實例一次一定效率較高。表 2.5 正說明了這個結果。

- 表 2.5 這裡列出了各種輸入大小，說明了在範例 2.8 中，當 n 為偶數時門檻值為 64，當 n 為奇數時門檻值為 70。

n	n^2	$3\lceil \frac{n}{6} \rceil^2 + 16n$
62	3844	3875
63	3969	4080
64	4096	4096
65	4225	4307
68	4624	4556
69	4761	4779
70	4900	4795
71	5041	5024

2.8 何時不能使用 Divide-and-Conquer

- 一個大小為 n 的個體被分成兩個或更多大小接近 n 的個體。
 - 例如： $fib(n)=fib(n-1)+fib(n-2)$
- 一個大小為 n 的個體被分成 n 個大小為 n/c 的個體，其中 c 為常數。
 - 例如：Towers of Hanoi Problem
 - [Play Tower of Hanoi \(mathsisfun.com\)](http://mathsisfun.com)
 - [Tower of Hanoi - Maths Careers](#)