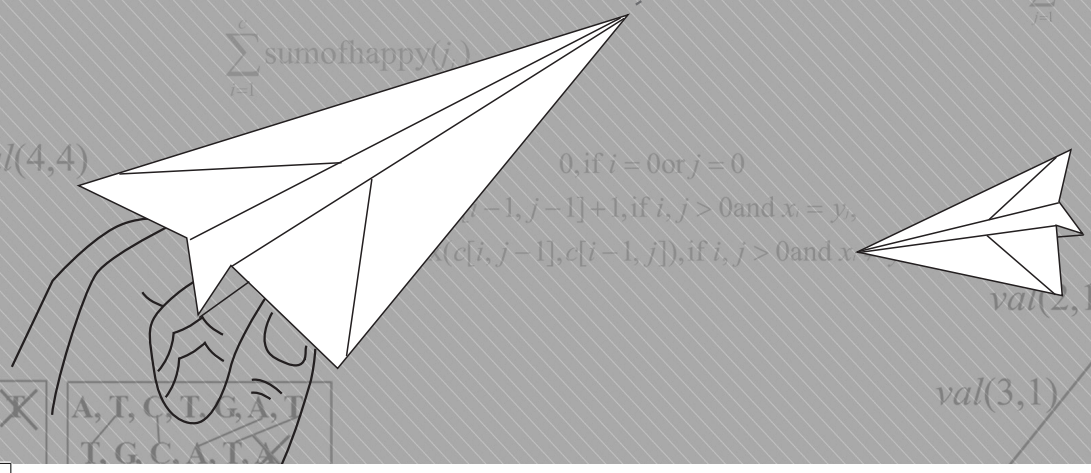
2 各個擊破法

章節大綱

「鄧哀王沖，字倉舒。少聰察岐嶷，生五六歲，智意所及，有若成人之智。時孫權曾致巨象，太祖曹操欲知其斤重，訪之群下，咸莫能出其理。沖曰：『置象大船之上，而刻其水痕所至，稱物以載之，則校可知矣。』太祖大悅，即施行焉。」

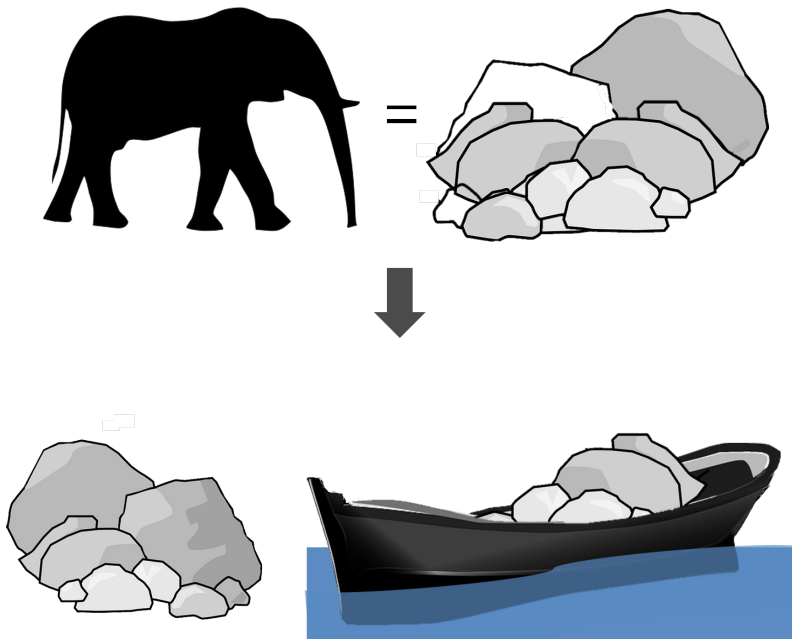
三國志魏書曹沖傳



2.1 何謂各個擊破法？

「何謂各個擊破法 (divide and conquer)？」

「簡言之，將一個問題切割(divide)成一些小問題，並且遞迴地解決(conquer)後，再利用這些小問題的解，合併(merge)成原來大問題的解。」



古時候要直接秤大象的重量不易，於是利用等重且較小的物品來取代量測，分次秤出所有取代的物品後，再加總即可得大象的重量。古人曹沖秤象的典故，其實蘊藏各個擊破法的精神。在本章我們將利用幾個例子，來介紹這個相當常見的演算法技巧。

2.2 最大值問題

第一個問題是在一個整數的集合中，找到最大值。如下表所示。

表 2.1 最大值問題

問題	在 *n* 個數字中找出最大值
輸入	任意的 *n* 個數字不規律地儲存於 *A* 矩陣中 4、2、17、5、22、8、13、6
輸出	此 *n* 個數字中最大值 22

最大值問題(finding the maximum)可使用一個迴圈，便可輕易解決。如下列的演算法所示。

表 2.2 最大值問題的演算法

輸入	*n* 個數字儲存於 *A* [1:*n*]中
輸出	*max* : *n* 個數字中的最大值

步驟	Step 1: 設定第一個元素為最大值。即 *max* = *A*[1]。
	Step 2: 執行迴圈 For *i*=2 to *n* do
	{
	If (*A*[*i*] > *max*) then *max* = *A*[*i*]。
	}
	Step 3: 輸出 *max*。

我們以最大值問題來介紹第一個各個擊破的演算法。如果要在一堆數字中找到最大值，可以將這些數字分成平均的兩堆。分別從這兩堆數字中找到其最大值後，比較這兩個值，找出大者便可得解 (圖 2.1)。

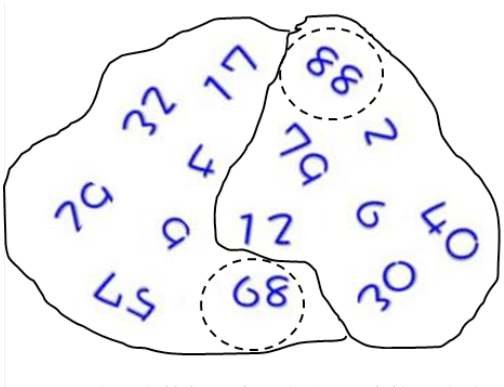


圖 2.1 將一堆整數分成兩堆後，再來找最大值

若將 100 筆資料存放在矩陣 $A[1:100]$ 中，首先將此矩陣從中間分成兩個均等的小矩陣 $A[1:50]$ 及 $A[51:100]$ (此為切割(divide)步驟)。若我們可以在 $A[1:50]$ 中找到最大值，而且在 $A[51:100]$ 中找到另一個最大值。最後，從這兩數中找出大值 (此為 conquer 步驟)，即得到解答。利用這樣的觀念，第一個各個擊破的演算法就可被設計出來。

表 2.3 最大值問題的各個擊破演算法

輸入	*n* 個數字儲存於 *A*[*i* : *j*]
輸出	*Vmax*:*n* 個數字中的最大值

步驟	Procedure max-divide-and-conquer (*i*, *j*, *Vmax*)
	{
	Step 1: if *i*=*j* then *Vmax*=*A*(*i*); 並輸出 *Vmax* ; break。
	Step 2: if *i*=*j*-1 then *Vmax*=*max*(*A*[*i*], *A*[*j*]); 並輸出 *Vmax* ; break。
	Step 3: 計算矩陣中間位置; 即 *mid*=(*i*+*j*)/2。
	Step 4: 遞迴呼叫找出一半矩陣中的最大值; 即 max-divide-and-conquer(*i*, *mid*, *Vmax1*)。
	Step 5: 遞迴呼叫找出另一半矩陣中的最大值; 即 max-divide-and-conquer(*mid*+1, *j*, *Vmax2*)。
	Step 6: 選出 *Vmax1* 及 *Vmax2* 的大值; 即 *Vmax*=*max*(*Vmax1*, *Vmax2*)並輸出 *Vmax*。
	}

2.3 時間複雜度

「上節的兩個演算法，哪一個比較好？」

「怎樣評斷一個演算法的好或壞？」

「比方說：擁有較少的程式執行時間，也就是較小的時間複雜度(time complexity)。」

「什麼是演算法的時間複雜度？」

「簡言之，就是一個演算法要花多少時間執行。」

以最大值問題的演算法(表 2.2)為例。每個指令被執行的次數羅列如下:

表 2.4 最大值問題的演算法時間複雜度

最大值問題演算法的指令	執行次數
max=*A*(1)	1
For *i*=2 to *n* do	*n*
If *A*(*i*)>*max* then *max*= *A*(*i*)	*n*-1
Output *max*	1
執行指令總數	2*n*+1

這個演算法的時間複雜度為 $2n+1$，此 n 是輸入數值的個數。因為 $2n+1$ 是 n 的兩倍(常數倍)加一，當 n 夠大時, $2n$ 和 $3n$、$5n$ 差不多和 n 成正比， 因此我們常用 $O(n)$ (讀作 big O n) 來簡稱此演算法的時間複雜度。下面是 O 的正式定義。

表 2.5 O 的正式定義

符號	O
定義	$O(g(n))$代表一函數的集合$\{f(n)$:存在一個正常數 c 和 n_0 使得所有的 $n \geq n_0$ 時 $0 \leq f(n) \leq cg(n)\}$
範例	$3n=O(n)$ $100n=O(n)$ $300n+100=O(n)$ $300n^2-40n+50=O(n^2)$
使用時機	利用 $O(g(n))$將演算法分成不同的層級

根據 O 的定義，表 2.2 上的最大值問題的演算法有 $2n+1=O(n)$時間複雜度。相對地，最大值問題的各個擊破演算法(表 2.3)的時間複雜度為多少呢?令執行 Procedure max-divide-and-conquer(1, n, $Vmax$)所需的計算時間為 $T(n)$。因為此演算法將輸入的矩陣平均切割後，呼叫自己兩次，所以我們可以將 $T(n)$用兩個 $T(n/2)$取代後得 $T(1)=1$; $T(2)=2$; $T(n)=2T(n/2)+4$。解開此數學式子即可知 $T(n)$。如何知道此數學式子的解呢?最簡單的方法是用「**猜**」。

表 2.6 數學式子 $T(1)=1$; $T(2)=2$; $T(n)= 2T(n/2)+4$ 的值

n	1	2	4	8	16	32	64
***T*(*n*)**	1	2	8	20	44	92	188

直覺上看起來 $T(n)$都小於 n 的常數倍，故 $T(n)$可能為 $O(n)$。若以 O 的角度分類，則此兩個演算法可歸類於 $O(n)$這一個等級的演算法。

2.4 二維極點問題

若將上節的最大值問題延伸到座標平面上，便成為二維極點問題(the 2-dimensional maxima finding)。

一個平面上的點的座標可用兩個整數(x, y)來表示。當 $x_1>x_2$ 而且 $y_1>y_2$ 時，我們說一個平面點(x_1, y_1)***支配***(dominate)另一個點(x_2, y_2)。簡言之，右上方的點支配左下方的點。

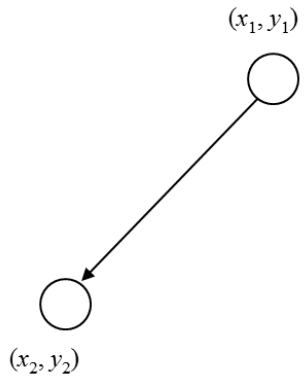


圖 2.2 平面點(x_1, y_1)支配另一個平面點(x_2, y_2)

二維極點問題就是在一些平面上的點中找到那些沒有被其他點支配的點，即是***極點***(maxima)。直覺上，極點就是所有點中，位居右上方的一些點。如下表和下圖所示。

表 2.7 二維極點問題

問題	在平面上的 n 個點中，找到那些沒有被其他點支配的點
輸入	*n* 個平面上的點 (2, 4)、(3, 10)、(5, 3)、(6, 8)、(8, 2) (10, 6)、(13, 5)、(15, 7)
輸出	找到所有沒有被其他點支配的點(極點) (3, 10)、(6, 8)、(15, 7)

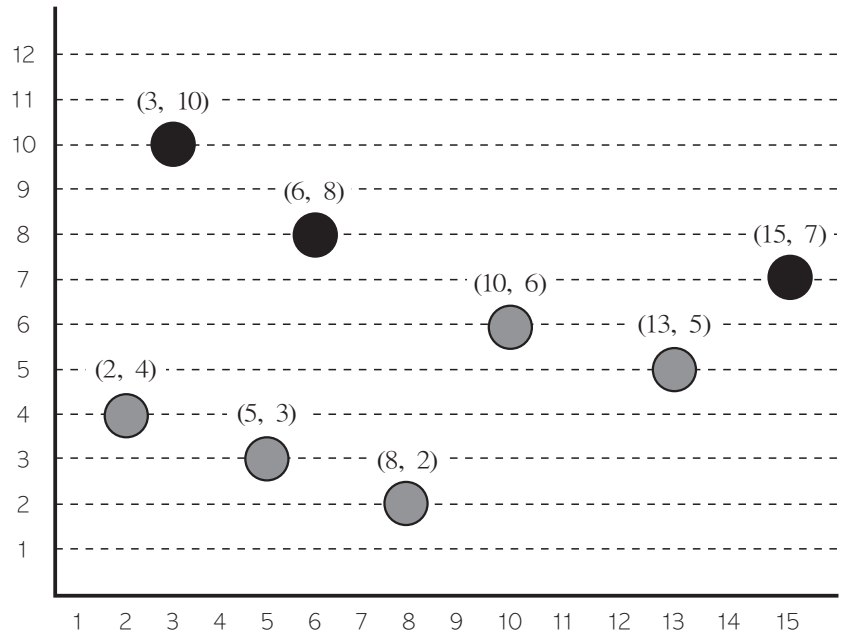


圖 2.3 二維極點問題是找出所有極點(圖中深黑色的點)

解決二維極點問題的最簡單方法是採用***暴力法***(brute force method)。想法很簡單，若一點 A 其右上方有一點 B，則 A 必不是極點。因為極點的右上方一定沒有其他點，如下圖所示。

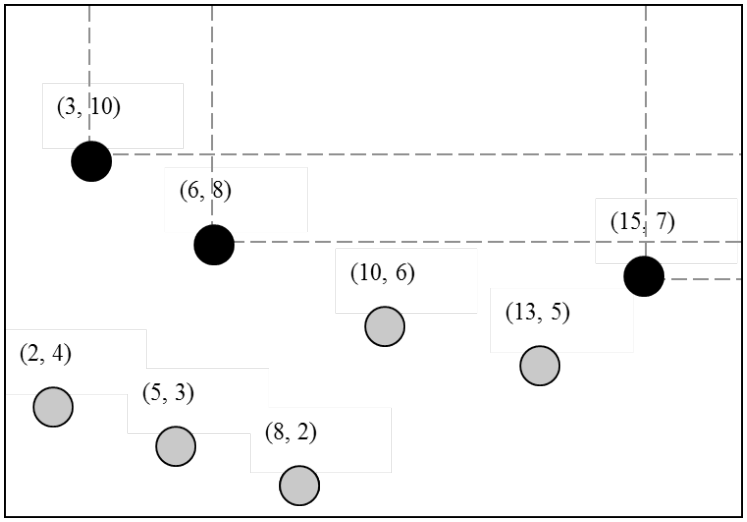


圖 2.4 極點(深黑的點)的右上方必沒有其他點

此暴力法即是每一點 A 都和其他的點 B 比較。若 B 在 A 的右上方，則排除 A 為極點的可能。最後留下(未被排除)的點即為極點。例如在圖 2.4 中，比較(8, 2)和(10, 6)兩點後，捨棄(8, 2)。(10, 6)並非極點，因為(10, 6)在和(15, 7)比較後，也被捨棄。此方法的時間複雜度取決於兩點比較的總次數。

「若給 n 個點，任意兩點需要比較一次，總共要比幾次？」

「從 n 個點中，任意選兩點的所有組合，也就是 $C(n, 2)$或 $\binom{n}{2}=\frac{n\times(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}=O(n^2)$。」

這個暴力法雖然解決了二維極點問題。但是，我們仍想知道是不是有其他更快的方法?

「這個問題可以利用各個擊破法來解決？」

「但.....什麼是各個擊破法？」

「將一個問題切割(divide)成一些小問題並且遞迴地解決後，再利用這些小問題的解合併成原來問題的解。」「你認為設計各個擊破法的第一步應先思考什麼？」

「如何將二維極點問題作切割吧？」

「怎樣切割呢？」「最簡單的切割方法是什麼？」

「從中間割成兩半吧!」

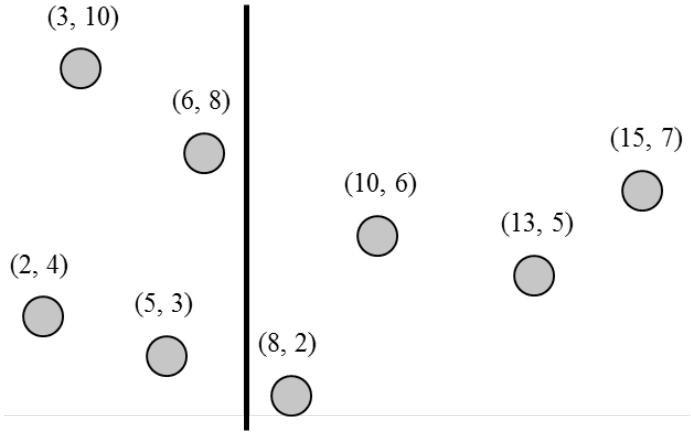


圖 2.5 將平面的點切成左右均等的兩半

將平面的點切成左右均等的兩半後(圖 2.5)，分別(遞迴地)找到左邊點的極點集合及右邊點的極點集合後，再將兩組的極點合併成最後的解(圖 2.6)。

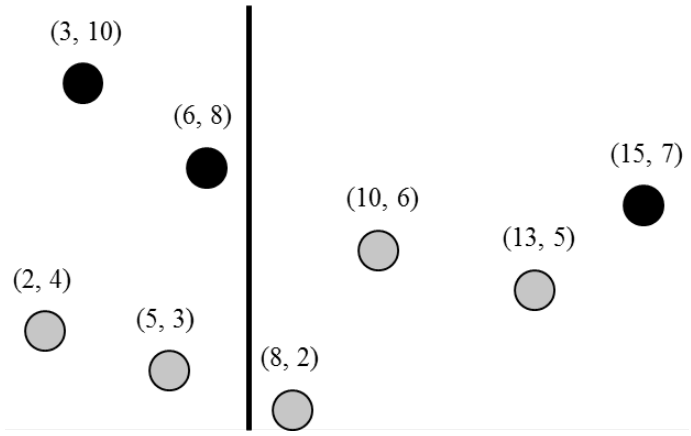


圖 2.6 先找到左半邊的極點及右半邊的極點

在圖 2.6 中欲合併左右兩邊的極點，好像只需要將兩集合聯集就好了。再多試幾個例子後，將會發現將左半邊的極點，和右半邊的極點直接聯集後，不一定是最後的解(圖 2.7)。原因是在聯集中出現有問題的點(圖 2.7 中打 × 的點)。

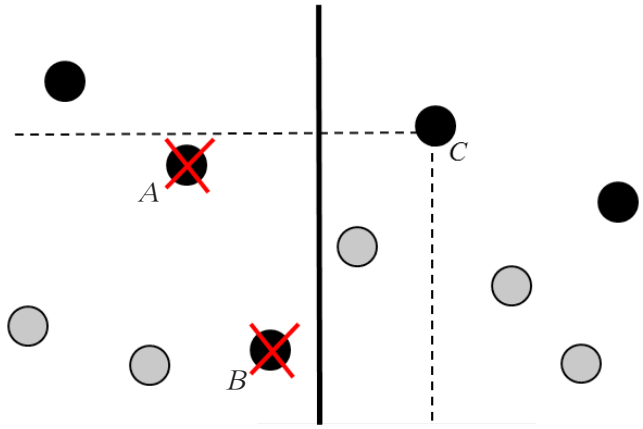


圖 2.7 當左半邊的極點被右半邊的極點所支配時，必不是極點

有趣的是，有問題的點都在左半邊，原因是有一些在左半邊找到的極點，有可能被右半邊的極點所支配著(圖 2.7 中 A 及 B 為 C 所支配)。注意右半邊點的 x 座標必大於左半邊點的 x 座標，故當右半邊的點其 y 座標也大於左半邊點的 y 座標時，右邊的點便支配這左邊的點，這些左半邊的點必不是極點。

因此令右邊的極點集合中，最高(或最左)的極點為 C。則當合併時，左邊極點集合中其高度小於 C 的點必不是極點(因為這些點必為 C 所支配)。如圖 2.7 中的 A、B 就必須被除去，因為這兩點被 C 所支配。最後合併的解答如圖 2.8 所示。此各個擊破的演算法羅列於表 2.8 中。

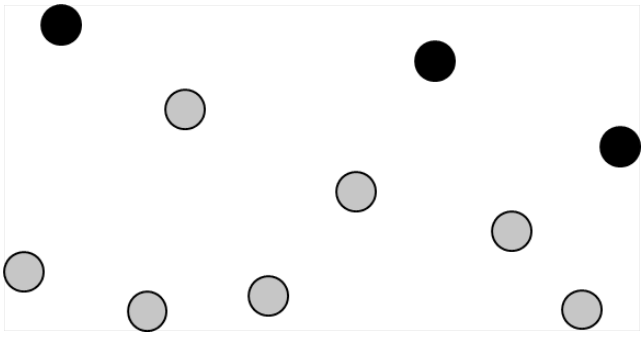


圖 2.8 合併後的極點

表 2.8 二維極點問題的各個擊破演算法

輸入	n 個平面上的點
輸出	極點集合 U
步驟	Algorithm two-dim-maxima {/*初次執行時將 n 個平面上的點按照 x 座標排序。*/ Step 1: 如果輸入只有一個點(即 n=1)，則回傳此點並結束。 Step 2: 將 n 個平面上的點切割成左右兩個集合 L 和 R。 Step 3: 遞迴地找出 L 和 R 的極點集合。 /*即遞迴呼叫 two-dim-maxima 兩次*/ Step 4: 找出 R 的極點集合中擁有 x 座標最小值 C。 Step 5: L' 為極點集合 L 中去除 y 座標小於 C 的點。 Step 6: 合併 L' 和 R 成為極點集合 U。 }

二維極點問題的各個擊破演算法的時間複雜度是多少？令 $T(n)$為當輸入的 n 點時所需的執行時間，則 $T(n)$大約為 $2T(n/2)+O(n)$。若解開此遞迴式，可得 $T(n)=O(n \log n)$。

2.5 快速排序法

快速排序法(Quicksort)是另一個典型的各個擊破演算法。

「什麼是排序(sorting)問題？」

「簡言之，討論如何將資料由小排列到大的問題。」

下表對排序問題作基本描述。

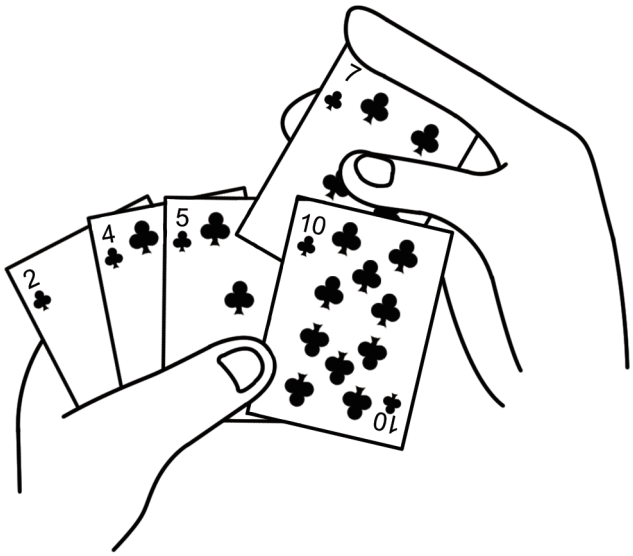
表 2.9 排序問題

問題	輸入 *n* 個數字，將這些數字由小到大排列好
輸入	任意大小的 *n* 個數字 65、70、75、80、85、60、55、50、45
輸出	此 *n* 個數字由小到大的順序 45、50、55、60、65、70、75、80、85

以下舉一個例子來說明快速排序法。

「快速排序法的基本技巧是什麼？」

「切割(partition)：利用一個數，將所有其他的數值分成左右兩邊，使得左邊的數都小於或等於此數；而且，右邊的數都大於或等於此數。」



下面的例子中，先將九個整數存在陣列中。取矩陣中第一數，當作***切割元素***(partitioning element)(即圖 2.9 中的 65)。並於矩陣中第二數開始向右尋找 ≥65 的數，同時於矩陣最後一數向左尋找 ≤65 的數。交換此兩數的位置，重覆前述動作，直到左右兩方的找尋箭頭交叉，此時將切割元素與所指到比它小的數交換位置。注意，圖 2.9 中數字下的箭頭 → 或 ← 代表目前找尋的方向及位置。

```
(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)  i  p
65  70  75  80  85  60  55  50  45  +∞    2  9     /* 65 為切割元素*/
    →                               ←
65  45  75  80  85  60  55  50  70  +∞    3  8     /*70 與 45 交換後*/
        →                   ←
65  45  50  80  85  60  55  75  70  +∞    4  7     /*75 與 50 交換後*/
            →           ←
65  45  50  55  85  60  80  75  70  +∞    5  6     /*80 與 55 交換後*/
                →   ←
65  45  50  55  60  85  80  75  70  +∞    6  5     /*85 與 60 交換後*/
                ←   →
65  45  50  55  60  85  80  75  70  +∞    end      /*小數 60 與 65 交換*/
60  45  50  55  65  85  80  75  70  +∞    end
```

圖 2.9 切割為快速排序法的主要步驟

切割演算法在掃描整個矩陣後，有一處疑點需要簡單討論。

「為何要將切割元素與所指到比它小的數交換位置？」

「如果將切割元素與所指到比它大的數交換位置，會怎樣呢？」

「比切割元素大的數將會被置於矩陣第一位置，這樣會造成有比切割元素大的數被置於左邊，如此就違背原意：利用切割元素，將所有其他的數值分成左右兩邊，使得左邊的數都小於或等於此數；而且，右邊的數都大於或等於此數。」

表 2.10 即為此切割演算法。

表 2.10 切割演算法

輸入	$a[m], a[m+1], \cdots, a[p-1]$。其中切割元素 $t = a[m]$
輸出	將切割元素 t 置於 a 矩陣的適當位置 $a[q]$，使得 $a[m]$到 $a[q-1]$的值都小於或等於 t；而且 $a[q+1]$到 $a[p-1]$的值都大於或等於 t

步驟	Algorithm partition(a, m, p)
	{
	  Step 1: v:=a[m]; i:=m; j:=p;
	  /*令 v 為切割元素，i 為矩陣開始的註標，而 j 為矩陣結束的註標。*/
	  Step 2: 重覆以下步驟，直到結束。
	  {
	  repeat
	  i:=i+1;
	  until (a[i] >=v); /*找到比切割元素大之數*/
	
	  repeat
	  j:=j-1;
	  until (a[j] <=v); /*找到比切割元素小之數*/
	
	  if (i<j) then 交換 (a[i], a[j]);
	  }
	  until (i>=j); /*以上兩方找尋的註標交錯*/
	  a[m]:=a[j]; a[j]:=v; return j;
	  /*將切割元素置於 a[j]處並且回傳j*/
	}

利用切割演算法，可以設計出快速排序法的演算法(表 2.11)。

表 2.11 快速排序法的演算法

輸入	$a[p], \cdots, a[q]$
輸出	$a[p], \cdots, a[q]$ 依照由小到大的數值順序排好

步驟	Algorithm quicksort (p, q)
	{
	  if (p<q) then
	  {
	  j:=partition(a, p, q+1); /*執行切割演算法(表 2.10)*/
	  /*切割元素所在的陣列註標被回傳並儲存於變數j中*/
	
	  /*以下兩個副程式的呼叫遞迴地排序兩個小矩陣的數值*/
	  quicksort(p, j-1);
	  quicksort(j+1, q);
	  }
	}

2.6 快速排序法的時間複雜度

「快速排序法的時間複雜度是多少？」

當輸入 n 個數字時，令快速排序法所需的執行時間為 $T(n)$。利用切割元素會將整個矩陣(n 筆資料)切割為兩個未排序的小矩陣，之後再遞迴地呼叫兩次快速排序法，來處理剩下的排序工作。

根據上述的說明，我們可得 $T(n)=T(j)+T(n-j-1)+n+1$。此處 $T(j)$及$T(n-j-1)$為兩個小矩陣，被遞迴呼叫兩次快速排序法所需的時間，而 $n+1$ 是切割演算法掃描矩陣(直到箭頭交錯)所需的比較(comparison)時間(圖 2.9)。為了簡化此分析，在此我們僅考慮所需的比較(comparison)時間，並不包含交換(exchange)所需的時間。

若能解開此遞迴式，即可得快速排序法的時間複雜度 $T(n)$。顯然地 $T(n)$之值可能會因 j 值的不同，而有所變化。以下，我們將討論快速排序法的最佳、最差及平均時間複雜度。

2.6.1 快速排序法的最佳時間複雜度

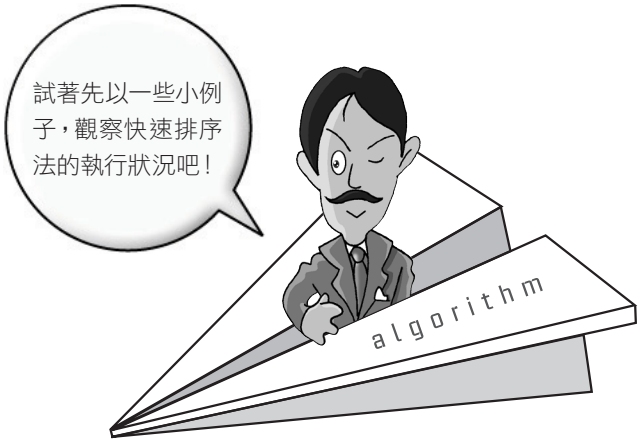
「在何種狀況下，快速排序法執行最快？」

「先想想什麼是快速排序法執行所需的時間？」

「即遞迴式 $T(n)=T(j)+T(n-j-1)+n+1$ 之解。」

「當 j 為何值時，$T(n)$會是最小？」

「可能是 $j=1$ 或 n 或 $j=\lfloor n/2 \rfloor$ 或 $\lceil n/2 \rceil$。」



顯然，上述討論的答案取決於以下兩式的解:

(1) $T(0)=0,\ T(1)=0,\ T(n)=T(\lfloor (n-1)/2 \rfloor)+T(\lceil (n-1)/2 \rceil)+n+1$

(2) $t(0)=0,\ t(1)=0,\ t(n)=t(1)+t(n\text{-}2)+n+1$

試著將此兩式 $T(n)$及 $t(n)$前幾項的值列出(表 2.12)。觀察此表可知，此兩式的值是隨 n 變大而變大(所謂遞增性質)。而另一個現象是，當 n 變得足夠大時($n \geq 6$)，$t(n)$的值總是大於 $T(n)$的值。

表 2.12 遞迴式 *T*(*n*)及 *t*(*n*) 前幾項的值

n	0	1	2	3	4	5	6	7	8	9	10	11	12
***T*(*n*)**	0	0	3	4	8	12	14	16	21	26	31	36	39
***t*(*n*)**	0	0	3	4	8	10	15	18	24	28	35	40	48

「最小值可能是 $T(n)$，最大值可能是 $t(n)$。」

「這表示在何種狀況下，快速排序法執行最快？」

「即 j 大約是 $n/2$ 時，會有最小值的 $T(n)$值，也就是快速排序法執行最快的時候。」

「這表示如何切割，快速排序法才會執行最快？」

「即平均切割，也就是切割元素位於正中間時。」

「我們知道了快速排序法在平均切割時，會有最佳的時間複雜度，但真正的理由是什麼？你可以一眼看穿嗎？」

「嗯…」

「或者，如何知道快速排序法最佳的時間複雜度？」

「可解開此遞迴式: $T(0)=0,\ T(1)=0,\ T(n)=T(\lfloor (n-1)/2 \rfloor)+T(\lceil (n-1)/2 \rceil)+n+1$。但好像有一點難！」

「您可想像有一個更容易的類似問題嗎？」

「有的。遞迴式: $T(0)=0, T(1)=0, T(n)=T(n/2)+T(n/2)+n+1=2T(n/2)+n+1$ 會比較簡單，尤其是當 $n=2^k$ 時。」

解開此簡單遞迴式的一個方法，即是利用重覆展開，如表 2.13 所示。

表 2.13 快速排序法最佳時間複雜度的近似解

$T(0)=0,\ T(1)=0,$
$T(n)=2T(n/2)+n+1,$ 當 $n=2^k$。
$=2(2T(n/4)+n/2+1)+n+1$ （因為 $T(n/2)=2T(n/4)+n/2+1$）
$=4T(n/4)+n+2+n+1$
$=4(2T(n/8)+n/4+1)+n+2+n+1$
$=8T(n/8)+n+4+n+2+n+1$
$=16T(n/16)+n+8+n+4+n+2+n+1$
.
.
.
$=2^k \times 0+\log_2 n \times n+(2^{k-1}+\ ...\ +4+2+1)$
$=\log_2 n \times n+(2^{k-1}+\ ...\ +4+2+1)$
$=\log_2 n \times n+n-1$
$=O(n\ \log\ n)$

Tip 注意 $T(0)=0,\ T(1)=0,\ T(n)=T(\lfloor (n-1)/2 \rfloor)+T(\lceil (n-1)/2 \rceil)+n+1$ 才是快速排序法的真正最佳的時間複雜度。然而其解可利用數學歸納法得之，而且也正好是 $O(n\ \log\ n)$。

2.6.2 快速排序法的最差時間複雜度

「快速排序法的最差時間複雜度是多少？」

「什麼是快速排序法執行所需的時間？」

「即此遞迴式 $T(n)=T(j)+T(n-j-1)+n+1$ 之解。」

「何時 $T(n)$會有最大值？」

「應該是遞迴式 $T(n)=T(0)+T(n-1)+n+1$ 之解。」

同樣地，疊代此遞迴式即可得快速排序法的最差時間複雜度 $O(n^2)$。

表 2.14 快速排序法最差的時間複雜度之解

$$
\begin{aligned}
&T(0)=0,\ T(1)=0,\\
&T(n)=T(0)+T(n-1)+n+1\\
&=0+T(n-1)+n+1\\
&=T(n-1)+n+1\\
&=(T(n-2)+n)+n+1,\ \text{因為 } T(n-1)=T(n-2)+n\\
&=T(n-2)+n+n+1\\
&=(T(n-3)+n-1)+n+n+1\\
&=T(n-3)+(n-1)+(n)+(n+1)\\
&\cdots\\
&=T(1)+3+4+\cdots+(n-1)+(n)+(n+1)\\
&=0+(3+n+1)(n-1)/2\\
&=(n+4)(n-1)/2\\
&=n^2/2+3n/2-2\\
&=O(n^2)
\end{aligned}
$$

2.6.3 快速排序法的平均時間複雜度

「最壞及最好的狀況不會常常出現才對呀？」「那一般平均的狀況，快速排序法的表現是如何？」「快速排序法的平均時間複雜度是多少？」

根據前兩小節的討論，快速排序法的執行速度和切割元素所在的位置，有密切的關係。簡單地說，當切割最平均時，快速排序法執行最快；相反的，當切割最不平均時，執行最慢。當輸入的數值是無規律時，這兩種狀況應不會每次都發生才對。倘若每一種切割出現的機率都是相同的(如圖 2.10 到圖 2.14 中↓所示的位置)，則快速排序法所需平均(期望)執行的時間是多少？

首先，令矩陣儲存共 n 個數值，當切割元素位於矩陣第一位時，其執行時間為 $T(n)=T(0)+T(n-1)+n+1$。

圖 2.10 切割元素落在矩陣的最左邊位置

當切割元素位於矩陣第二位時，其執行時間為 $T(n)=T(1)+T(n-2)+n+1$。

圖 2.11 切割元素落在矩陣的左邊第二個位置

當切割元素位於矩陣第三位時，其執行時間為 $T(n)=T(2)+T(n-3)+n+1$。

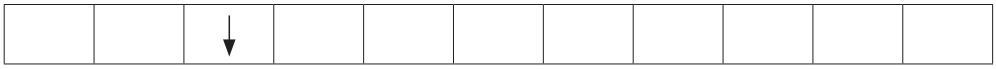


圖 2.12 切割元素落在矩陣的左邊第三個位置

以此類推，當切割元素位於矩陣最後第二位時，其執行時間為 $T(n)=T(n-2)+T(1)+n+1$。

圖 2.13 切割元素落在矩陣的右邊第二個位置

當切割元素位於矩陣最後第一位時，其執行時間為 $T(n)=T(n-1)+T(0)+n+1$。

圖 2.14 切割元素落在矩陣的最右邊位置

假設每一種狀況發生的機率，都是一樣的(即 $1/n$)。則快速排序法的平均時間複雜度，可將每一種狀況所需的執行時間加總後再除以 n，可得以下的遞迴式：

$$T(n)=((T(0)+T(n-1)+n+1)+(T(1)+T(n-2)+n+1)+\cdots+(T(n-2)+T(1)+n+1)+(T(n-1)+T(0)+n+1))/n$$

稍作整理一下可得：

$$T(n)=n+1+2/n\times(T(0)+T(1)+\ldots+T(n-1))$$

解開此遞迴式後，可得 $T(n)=O(n\log n)$。詳細的分析，請參考下表。

表 2.15 快速排序法的平均時間複雜度分析

$T(0)=T(1)=0$

$T(n)=n+1+2/n\times(T(0)+T(1)+\ldots+T(n-1))$

將上式等號的左右，各乘上 n 可得

$nT(n)=n(n+1)+2(T(0)+T(1)+\ldots+T(n-1))$。 ..... (A)

將上式所有的 n 用 $n-1$ 代入，得出另一等式

$(n-1)T(n-1)=(n-1)n+2(T(0)+T(1)+\ldots+T(n-2))$ ..... (B)

將(A)-(B)，可得一個較精簡的遞迴式：

$nT(n)-(n-1)T(n-1)=n(n+1)-n(n-1)+2(T(0)+T(1)+\ldots+T(n-1)-(T(0)+T(1)+\ldots+T(n-2)))$

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

整理一下可得

$nT(n)=(n+1)T(n-1)+2n$。

再將上面的等式，左右各除以 $n(n+1)$ 可得

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

疊代此遞迴式即可得

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

$$=\frac{T(n-2)}{n-1}+\frac{2}{n}+\frac{2}{n+1} \quad \text{因為} \quad \frac{T(n-1)}{(n)}=\frac{T(n-2)}{n-1}+\frac{2}{n}$$

$$=\frac{T(n-3)}{n-2}+\frac{2}{n-1}+\frac{2}{n}+\frac{2}{n+1} \quad \text{因為} \quad \frac{T(n-2)}{(n-1)}=\frac{T(n-3)}{n-2}+\frac{2}{n-1}$$

$$=\frac{T(1)}{2}+\frac{2}{3}+\frac{2}{4}+\ldots+\frac{2}{n+1}=\frac{T(1)}{2}+2\sum_{3\le k\le n+1}\frac{1}{k}=2\sum_{3\le k\le n+1}\frac{1}{k}$$

換言之，$T(n)=(n+1)\times 2\sum_{3\le k\le n+1}\frac{1}{k}$

而 $\sum_{3\le k\le n+1}\frac{1}{k}\le\int_{2}^{n+1}\frac{1}{x}dx=\log_e(n+1)-\log_e 2$

因此最後可得 $T(n)\le 2\times(n+1)\times[\log_e(n+1)-\log_e 2]=O(n\log n)$

2.7 找尋第 k 小值問題

最後一個例子是，在 n 個數中找尋第 k 小值問題(finding the k^{th} smallest value)(表 2.16)。當 k=1 或 n 時，就是找最小值或大值問題，在＜2.2 節＞中，我們利用一個迴圈便可輕易解決之。但當 k 不固定時，如何有效地解決此問題？

表 2.16 找尋第 *k* 小值問題

問題	輸入 *n* 個數，請輸出第 *k* 小值
輸入	*n* 個未排序好的數及 *k* *n* =8, 72, 34, 12, 84, 21, 45, 58, 63 *k*=4
輸出	第 *k* 小值 34 為第 4 小值

「如何在 n 個數中找尋第 k 小值？」

「可先將 n 個數由小到大排序 (sort)後，再從矩陣第 k 個位置取得。」

「如此所需的時間複雜度為多少？」

「若使用一般的排序法，需要 $O(n \log n)$。」

「有可能更快嗎？譬如說 $O(n)$」「如何在 $O(n)$時間內，找尋第 k 小值?」

「嗯…」

2.7.1 找尋第 k 小值的 $O(n)$演算法

本節的重點就是設計一個可以找到任意第 k 小值的 $O(n)$演算法。根據上述的討論，顯然不可以利用排序法(否則所需的時間將達到 $O(n \log n)$)。

此演算法需要的技巧源自於快速排序法中的切割演算法(參考＜2.5 節＞)。首先，利用快速排序法中的切割演算法，可將整個矩陣分成「比切割元素大」和 「比切割元素小」的兩個集合。之後，如下圖，切割元素 65 是第 5 小的數，若 k=5 則此數即為解；若 k>5 需找右邊集合，此時左邊集合可略過不必找；若 k<5 需找左邊集合，此時右邊集合可略過不必找。

遞迴地利用以上的步驟，即可找到第 k 小值。問題是，這樣的作法所需的最長時間等同於快速排序法的最差時間複雜度 $O(n^2)$。

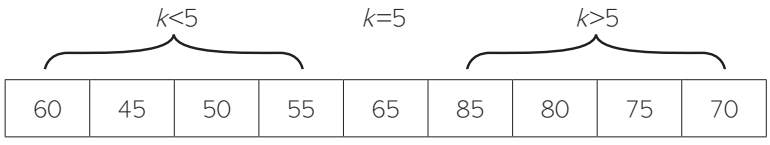


圖 2.15 切割元素的位置決定尋找第 k 小值的範圍

「上述方法的缺點是什麼？」

「應該還是使用各個擊破法時，切割元素並未平均切割整個矩陣。」

「未平均切割的原因何在？」

「因為切割元素有時不是太大就是太小。」

「最好的切割元素是什麼？」

「找中間的值當作切割元素。」

「如何找中間的值？」

「可先將 n 個數由小到大排序 (sorting)後，再從矩陣中間位置取得。」

「這樣的演算法可以在 $O(n)$內完成嗎？」

「糟了！不行！排序就花掉 $O(n \log n)$時間。」

「有其他方法可以較快地找中間的值？」「如果你一時不能解決此問題，可先嘗試一些相關的問題。你可以想像到一個更容易的相關問題嗎？」

「找到接近中間的值不知有沒有幫助？」

「如何很快地找到接近中間的值？」

「好像很難？」

「要找到中間的值，難在那裡？」

「好像這個值跟矩陣中每個數都有關？」

「怎樣的中間值比較好找？」

「如果從 5 個數中找中間值就容易多了。」

「這樣的方法，可以幫助您找到中間的值或較接近中間的值？」

「*n* 個數太多了，若是只有 5 個數就好找了。也許將 *n* 個數分成 5 個數一組一組…」

「然後呢？」

「將每一組的中間值找到。」

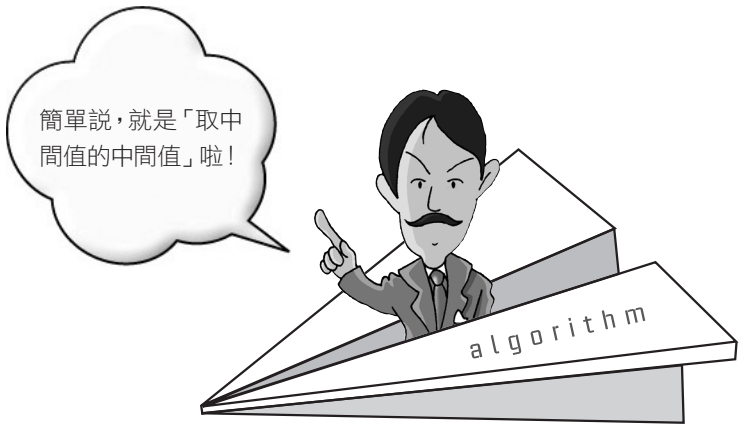
「共有幾組中間值？」「然後呢？」

「大約 *n*/5 組，然後再從這些值中找到其中間值。」

「這樣的值拿來當作切割元素，不知效果好不好？」

「嗯…」

以下的演算法，即是利用此想法，快速取得大致接近中間的值。例如，將 *n* 個數排列成每 5 個數一列，若最後一列不足則自成一列(如圖 2.16)。



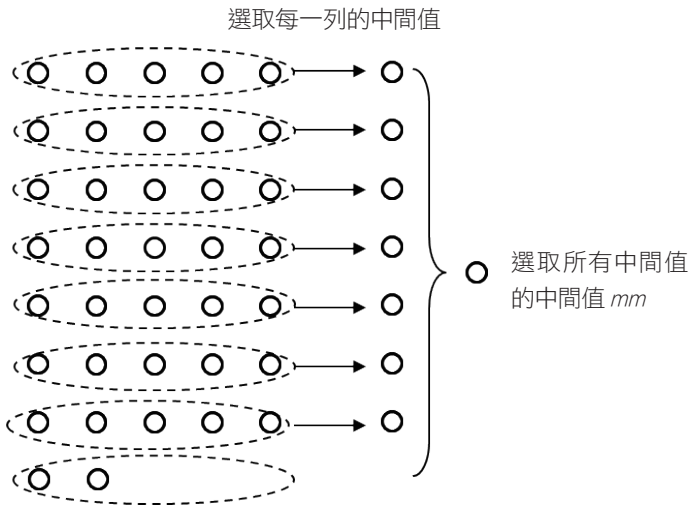


圖 2.16 將 n 個數排列成每 5 個數一列後，取其中間值之中間值

每列 5 個數分別找出其中間值(共有 $\left\lfloor \frac{n}{5} \right\rfloor$ 個數，如圖 2.16 所示)後，再從這些中間值中取出其中間值 mm。以下是利用 mm 當作切割元素來找尋第 k 小值的演算法。

表 2.17 找尋第 k 小值的 $O(n)$演算法

輸入	矩陣 $a[low:up]$
輸出	$a[low:up]$中第 k 小值

步驟	Algorithm selection (a, k, low, up)
	{
	Step 1: 計算矩陣中數值個數 n:=up-low+1;
	Step 2: if (n<=5) then 排序 $a[low:up]$並回傳第 k 小值;
	Step 3: 將 $a[low:up]$切割成 $\lceil n/5 \rceil$ 列;每列數目為 5(最後一列可能不足 5);
	Step 4: 將每列的中間值找出，並儲存於矩陣 $m[i](1 \le i \le \lceil n/5 \rceil)$;
	Step 5: 遞迴呼叫 selection 來找尋矩陣 $m[i]$中的中間值;即從 $\lceil n/5 \rceil$ 個 $m[i]$中找出第 $\left\lceil \lceil n/5 \rceil /2 \right\rceil$ 小值並存入 mm 中(即執行 mm:=Selection(m, $\left\lceil \lceil n/5 \rceil /2 \right\rceil$, 1, $\lceil n/5 \rceil$));
	Step 6: 利用 mm 當作切割元素，執行切割演算法 Partition(a, low, up) (表 2.10)後，回傳 mm 存於矩陣 a 的第 j 個位置(即註標為 j);
	Step 7: if (k=(j-low+1)) then 回傳第 k 小值為 $a[j]$;
	else if (k<(j-low+1)) then 第 k 小值為遞迴呼叫
	selection(a, k, low, j-1)後回傳的值;
	else 第 k 小值為遞迴呼叫 selection(a, k-(j-low+1), j+1, up)後回傳的值;
	}

Tip 注意從中間值中取出其中間值 *mm* 的步驟(即 Step 5)是遞迴地呼叫找尋第 *k* 小值的演算法，只是此時的 *k* 值是中間值總數的一半。

2.7.2 找尋第 *k* 小值演算法的時間複雜度分析

找尋第 *k* 小值的 $O(n)$演算法的關鍵是，所選的切割元素 *mm* 必須足夠靠近整個矩陣的中間值。如此可在切割後，無論向右找(比 *mm* 大的值)或向左找(比 *mm* 小的值)，都可保證至少有夠多(至少大約 $n/4$)的數不必找尋(圖 2.17)。

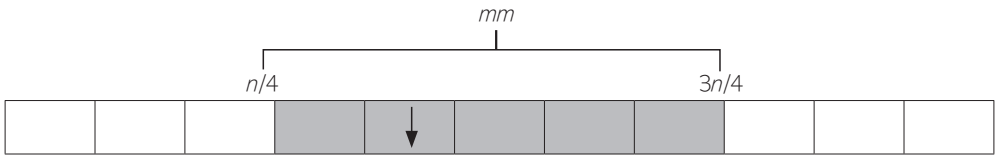


圖 2.17 切割元素 *mm* 為介於矩陣 *a* 第 $n/4$ 到第 $3n/4$ 小的數

以下說明切割元素 *mm* 屬於矩陣 *a* 中「大約介於第 $n/4$ 到第 $3n/4$ 小的數」的理由。因為 *mm* 是 $m[i]$ ($\lceil n/5 \rceil$ 列)中的中間值，故 $m[i]$ 中必有 $\lceil \lceil n/5 \rceil /2 \rceil$ 的值大於或等於 *mm*(如圖 2.18 中間橢圓形所示)。除了 *mm* 這列及可能未排滿的最後一列，在剩下的($\lceil \lceil n/5 \rceil /2 \rceil$-2)列中，每列有 $\lceil 5/2 \rceil$=3 的值大於或等於 $m[i]$(圖 2.18 右下長方形所示)。如此可得至少 $3\times(\lceil \lceil n/5 \rceil /2 \rceil -2) \geq \frac{3n}{10}-6$ 個數大於或等於 *mm*。

同理，也有多於 $\frac{3n}{10}-6$ 的值小於或等於 *mm*(圖 2.18 左上長方形所示)。因此在 Step 7 中(表 2.17)，遞迴呼叫 selection 時的輸入矩陣 *a* 中，最多有 $\frac{7n}{10}+6$(即 $n-(\frac{3n}{10}-6)$) (接近 $3n/4$)個數字。

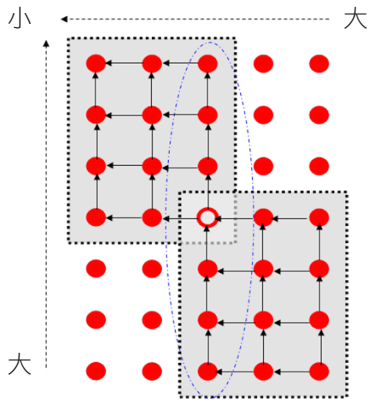


圖 2.18 至少 $\frac{3n}{10}-6$ 個數大於或等於 *mm*(右下長方塊)，同時至少 $\frac{3n}{10}-6$ 個數小於或等於 *mm*(左上長方塊)

Tip 注意圖 2.18 的數值排列是假想左上的值都小於右下的值。在演算法中，我們並不需要真正作此排列，畢竟此圖只是用來說明，切割元素 mm 的確足夠靠近整個矩陣的中間值。

表 2.17 中的步驟，除了 Step 5 和 Step 7 外，都可在 $O(n)$內完成。Step 5 需要 $T(\lceil n/5 \rceil)$時間而 Step 7 需要 $T(\frac{7n}{10}+6)$時間。因此找尋第 k 小值演算法的時間複雜度 $T(n)$便可用以下式子表示：

$$T(n) \le T(\lceil n/5 \rceil) + T(\frac{7n}{10}+6) + kn$$

此處的 k 為常數。利用數學歸納法可得(表 2.18)，當 $n>80$ 時，$T(n) \le cn$。

表 2.18 利用數學歸納法證明找尋第 *k* 小值演算法的時間複雜度 *T*(*n*)= *O*(*n*)

定理 2.1 $T(n) \le T(\lceil n/5 \rceil) + T(\frac{7n}{10}+6) + kn$ ，則 $T(n)<=cn$ 當 $n>80$。

證明：假設小於 n 時，本定理成立則

$$T(n) \le T(\lceil n/5 \rceil) + T(\frac{7n}{10}+6) + kn \le c(n/5+1) + c(\frac{7n}{10}+6) + kn$$

$$\le \frac{2cn}{10} + \frac{7cn}{10} + 7c + kn \le \frac{9cn}{10} + 7c + kn$$

$$\le cn$$

因為 $\frac{9cn}{10} + 7c + kn \le cn$ ，故 $7c + kn \le cn - \frac{9cn}{10} = \frac{cn}{10}$ 。

因此我們選擇 c，使得當 $n>80$ 時，$7c + kn \le \frac{cn}{10}$ 為真即可。

因此，我們設計出一個找尋第 k 小值問題的線性(即 $O(n)$)時間演算法，此為理論上最佳的演算法。

2.8 各個擊破法的技巧

「什麼問題可利用各個擊破法來解決呢?」

這個問題顯然不容易回答了。但是，一旦一個問題可被各個擊破法所解決，也常隱含此問題可被平行處理。最後，以下提醒您使用各個擊破法時的注意事項。

- **切割**(divide)：儘可能地快速地將問題平均地切割。若能將問題平均地切割成幾個小問題，常可降低所需的執行時間。
- **克服**(conquer)：利用遞迴解決切割後的小問題。
- **合併**(merge)：儘可能快速地合併上述小問題的解，成為原問題的解。合併時所需的時間也會影響到整個演算法所需的執行時間。

「還有什麼問題可被各個擊破法解決?」

以下列出一些可被各個擊破法解決的常見問題，供您參考。

1. **合併排序**(merge sort)：是一個利用矩陣中鄰近的數字兩兩合併，將數字由小到大排列的方法。
2. **二元搜尋法**(binary search)：利用一群已由小到大(或大到小)排序好數值來進行特定數值的搜尋。每次取出搜尋區間的中間數值，與欲搜尋的值作比較後，捨棄不可能的一半數值，並朝可能存放的另一半區域繼續尋找。直到尋得或確定不存在，方得停止搜尋。
3. **前置和**(prefix sum)：計算一維矩陣 B，使得 $B[i]$ 中儲存著從 A 矩陣中第一個值加到第 $i(1 \le i \le n)$個值的總和。此技巧常被用於設計平行處理的演算法。
4. **矩陣相乘問題**：將兩個矩陣快速相乘的問題。
5. **鄰近配對問題**(closest pair problem)：找出平面點中的最靠近的兩點的距離。

6. **凸殼問題**(convex hull problem)：找出一個包含所有輸入平面點的最小凸多邊形的問題。

7. **沃羅諾伊圖**(Voronoi diagrams)：一個將每一個平面點 x 分割落於單獨的一個區域，使得另一個新加的平面點 y 落於此區域時，其最近的點即為 x。此技巧可用於無線行動裝置 (y) 快速地尋找最近的基地台 (x)。

學習評量

1. 請設計一個各個擊破的程式來解決二維極點問題。

輸入：

```
8                    (平面點的個數)
2 4                  (以下是每個點的x和y座標)
3 10
5 3
6 8
8 2
10 6
13 5
15 7
```

輸出：

```
3  10
6  8
15 7
```

2. 請根據 O 的定義，證明當 $T(n)=T(n/2)+T(n/2)+1$時，$T(n)=O(n)$。

3. 請設計一個程式來執行快速排序法(quicksort)。

輸入：

```
65 70 75 80 85 60 55 50 45
```

輸出：

```
45 50 55 60 65 70 75 80 85
```