Ch6. 分治法

這章開始進入「三大演算法」的介紹，即分治法、貪婪演算法和動態規劃。

首先，要介紹的是其中的「分治法」。一樣會先簡介分治法是什麼，以及哪些問題適用分治法。

再來，會介紹幾個分治法的常見應用：

A. 河內塔 Hanoi Tower

B. 合併排序 Merge Sort

C. 快速排序 Quick Sort

D. 最大子數列問題 Maximum Subarray

E. 矩陣相乘 Matrix Multiplication

F. 選擇問題 Selection Problem

接著，會用「支配理論 Master theorem」來估計使用到分治法的算法複雜度。最後，會有幾題實戰練習。

1. 分治法 Divide and Conquer 簡介

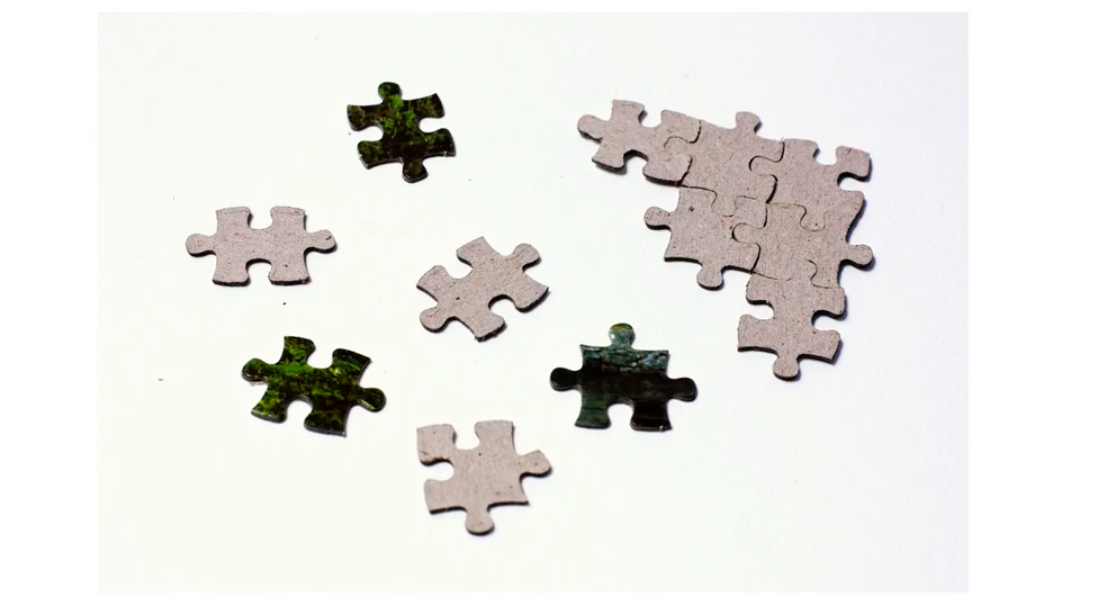
分治法實際上是一種程式設計的策略（Strategy），並沒有固定的 pseudo code 可供依循。後面兩章介紹的貪婪演算法、動態規劃也是像這樣的「策略」，或者看作解決問題時的「精神」。

這些策略雖然沒有被嚴格定義，但是學習它們背後的精神是很重要的。在學習的過程中，要時時問自己：為什麼這個策略要這樣做？這樣做可以包含所有答案的可能性嗎（必須要都有包含才可以採取該策略，否則會出錯）？並且，要有能力估計採取特定策略解決問題時複雜度為何。

分治法的英文是 Divide and Conquer，顧名思義，先「切割」再「征服」問題，這代表先把原本的一個「大問題」切割成數個「小問題後」後，再分別解決。

一般來說，分治法中的大問題和切割得到的許多小問題解法是「一樣的」，只有輸入資料大小不同，因此可以利用「遞迴」的運作特性來解決。通常，在將問題縮小到一定的規模以下後，解答會顯而易見，這時再把這些小問題的答案合併，就可以進一步得到原問題的答案。

分治法的精神類似孫子兵法中的「親而離之」，一個大問題或許不容易解決，但是一旦將其拆解，分個擊破，那麼最後大問題也迎刃而解。



拼拼圖的過程中，一開始同樣有成千上百片的拼圖，不可能一眼就看出該如何完成，但是每次給定兩片拼圖，決定它們是否能夠拼在一起，以及可以時，應該如何拼在一起，那麼就可以一步一步完成整個拼圖。

（1）迭代與遞迴

迭代通常使用 for 來跑遍整個範圍，比如下面的程式碼中，sum 是由迴圈中一層一層加上新的數字產生。

|  |  |
| --- | --- |
| 迭代求和 | |
| 1  2  3  4  5  6 | int sum(int n){  int sum = 0;  for (int i=1 ; i<=n ; i++)  sum += i;  return sum;  } |

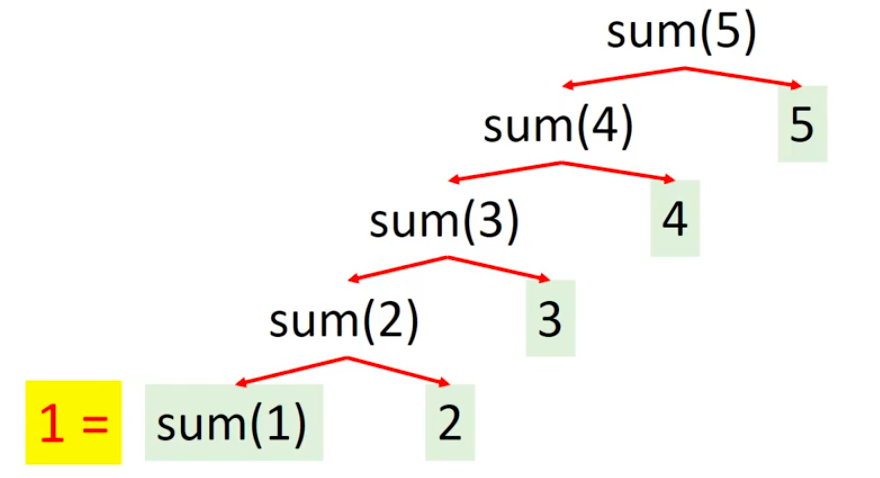
|  |  |
| --- | --- |
| 迭代求積 | |
| 1  2  3  4  5  6 | int factorial(int n){  int result = 1;  for (int i=1 ; i<=n ; i++)  result \*= i;  return result;  } |

相對的，遞迴則是簡化步驟後，透過在函式中再呼叫函式本身來解決問題：

--- （1）

--- （2）

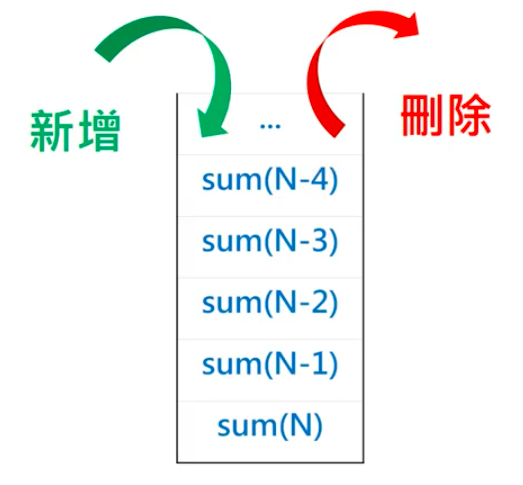
因此，

像上面的幾個式子所示， 可以由 和 得到，類似的， 也可以由 和 得到，不斷推下去，當到達 時，就可以馬上得到答案是 。

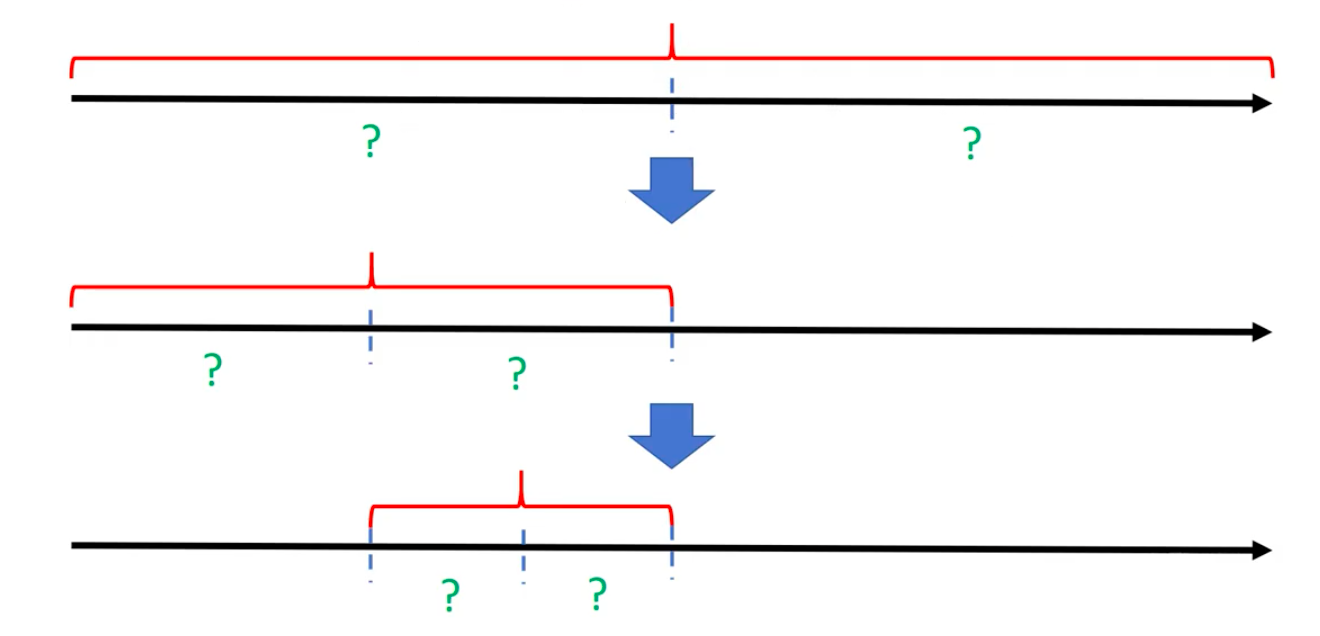
|  |  |
| --- | --- |
| 遞迴求和 | |
| 1  2  3  4  5  6 | int sum(int n){  if (n==1)  return 1;  else  return sum(n-1)+n;  } |

因此遞迴分成兩個部分：第一個是「怎麼化簡問題」；第二個是「化簡到哪裡可以停」，也就是「結束條件」。

比如圖中 sum(5) 被一路化簡，當到達 sum(2) 時，將其視為 sum(1) 和 2 的和，但是 sum(1) 不應再被化簡下去，而應該直接取其值為 1。

遞迴的優點是其運作簡潔、直觀，但是缺點是空間效率較差，每次呼叫函式時，都必須為函式配置記憶體，重複呼叫一百次，就要在某個時間點同時使用到一百個函式的記憶體空間。迴圈則與此不同，每次進行迴圈時，都可以把前幾次進行使用到的局部變數 local variable 釋放掉，因此不會有執行過程中佔用空間越來越大的情形發生。通常，如果可以做到，都要考慮將遞迴重構為迴圈。

另外，遞迴的執行過程中需要用到堆疊 stack 資料結構，因為 stack 的容量有上限，遞迴的呼叫次數也有上限（超過會產生溢位，也就是 stack overflow），相對的，迴圈的執行次數則通常沒有上限。

（2）重訪二分搜尋法

前面已經學過的二分搜尋法其實也是分治法的應用，每次搜尋時，都把下界到上界分成兩段，看是在左邊還是右邊，當知道在哪一邊後，要搜尋的範圍就變成原本的一半。

二分搜尋法可以視為一種特化的分治法：

A. 把母問題切割成兩個子問題（左邊和右邊）--- Divide

B. 使用同樣的算法或函式處理 --- Conquer

C. 把每個小問題的答案合併成母問題答案 --- Combine

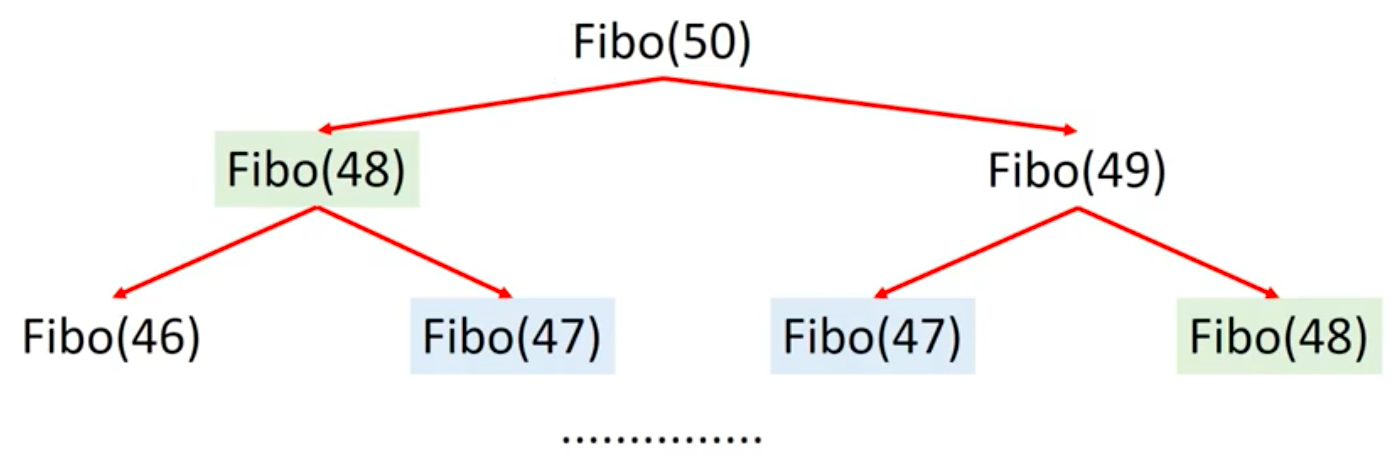
不過二分搜尋法比較特別的是每次切割出兩個子問題後，只處理其中一個，另一個則因為已經確定答案不在其中，不做處理。

（3）費波那契數列

另一個例子是費波那契數列，費波那契數列的定義是其前兩項為 1，之後每一項都是前面兩項的和，數列如下：

|  |  |
| --- | --- |
| 用分治法生成費波那契數列 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | #include <iostream>  using namespace std;  int Fibo (int n){  if (n<=2)  return 1;  else  return Fibo(n-1) + Fibo(n-2);  }  int main()  {  // 求費波那契數列的前一百項  for (int i=1 ; i<=100 ; i++){  cout << Fibo(i) << " ";  }  return 0;  } |

實際執行看看，會發現隨著 變大，需要的計算時間顯著增加，這似乎有點令人困惑，每一項只是前兩項相加而已，既然之前已經算過前兩項的值了，再執行一次加法為什麼會耗費這麼多時間呢？



這是因為每次計算得到的結果都「沒有被儲存下來」，舉例來說，計算 的時候，要由 和 的和決定，左邊的子樹中計算了一次 ，右邊的子樹中為了計算 ，也獨立算了一次 ，結果就是要得到每一項，都要從 和 一路往上加，進行了大量重複的運算。

另外，從樹的結構判斷，一個 被拆解成兩項，這兩項又各被拆解成兩項，這樣一來需要計算的次數以 2 的次方速度增加，會是複雜度很糟糕的演算法，代表費波那契數列其實不適合簡單使用分治法來處理。

如果用迭代（迴圈）來處理費波那契數列，會宣告出一個陣列 array，每次算出一個費波那契數，就放到陣列中以利後續使用，不用再重新計算，這個方法就是後面章節會提到的「動態規劃」。

|  |  |
| --- | --- |
| 用迭代產生費波那契數列 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | int fibo(int n){  // 宣告儲存每個費波那契數的陣列  int \*data = (int\*) malloc(sizeof(int)\*n);  data[0] = 1;  data[1] = 1;  // 每個費波那契數是用兩個先前已經儲存的結果直接求和得到  for (int i=2 ; i<n ; i++){  data[i] = data[i-1] + data[i-2];  }  int result = data[n-1];  free(data);  return result;  } |

兩種寫法相較，迭代的效率較佳，但是架構不清楚，相對的，遞迴則是效率差，但架構清楚，程式碼比較簡潔。

（4）不適合使用分治法的情境

從上面費波那契數列的例子，就可以看出什麼樣的情形不適合使用分治法：當問題會被切割成兩個以上的子問題時，需要的計算量就會呈指數增加。假設每次拆解成 個子問題、拆解 次，那麼最後就會變成 ，呼叫這麼多次函式通常是不能接受的。

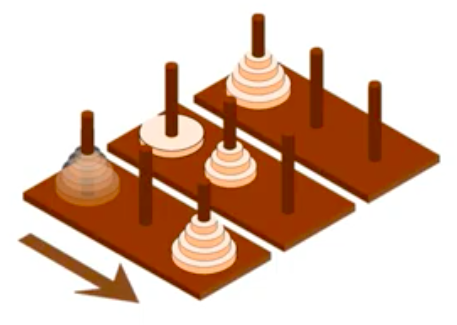
然而，也有一些情況下指數成長本來就無法避免，比如馬上會介紹到的「河內塔」的例子，因為限定一次只能搬動一個圓盤，所以沒有辦法進一步化簡，需要處理的次數一定會呈指數增加。

（5）分治法適用的小結

可能適用分治法的情形：問題可以被切割成「單一」、「同樣」的子問題。

可能不適用的情形：問題會被切割成「兩個以上」的子問題，視情況而定，可能要嘗試改用動態規劃。

2. 河內塔

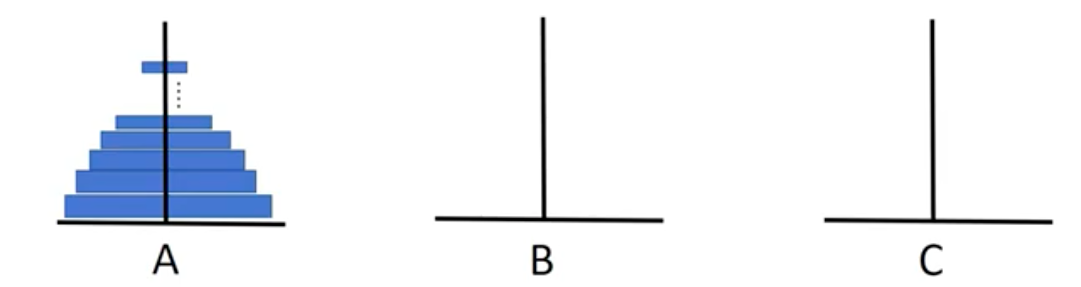
接下來，會介紹幾種分治法的常見應用，首先是「河內塔」。

河內塔是一種古老的遊戲，遊戲的規則可以表示為：

A. 共有三根棍子，每根棍子上可以擺放圓盤。

B. 所有棍子上，較上方的盤子都必須較下方的「所有盤子」來得小，也就是說，對於每根棍子而言，只要上面有盤子，盤子就必須是從底層開始由大到小依序擺放。

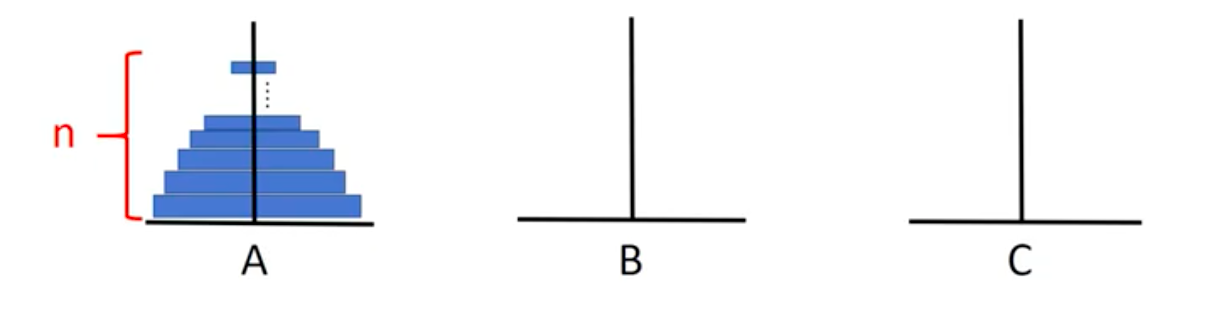
C. 圓盤開始至完成間，每次只能移動一個盤子。



今天有 個圓盤，目標是要把所有盤子從 A 棍子移到 C 棍子上，總共要移動幾次、怎麼移動呢？

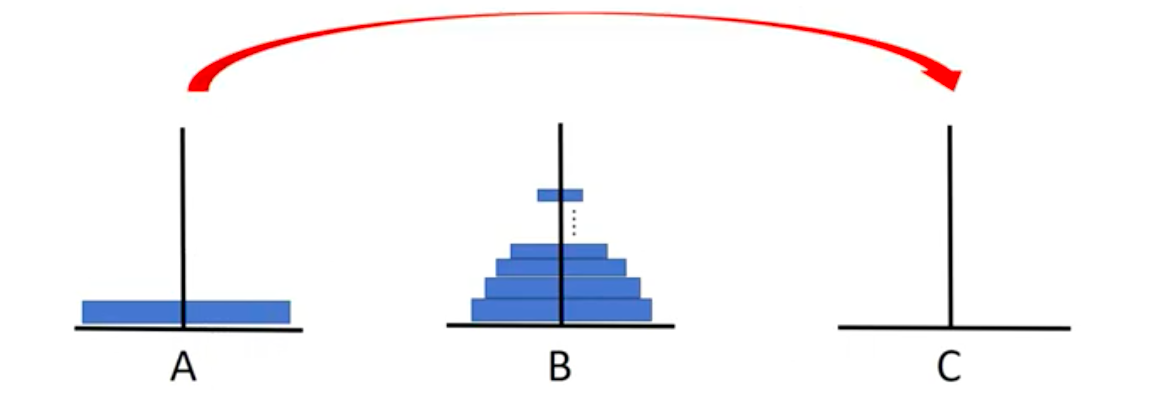
（1）用分治法解決河內塔問題

觀察這個問題，因為大的盤子一定要放在小盤子的下面，所以最下面 / 最大的盤子必須先移動到目標的棍子上。現在 A 棍上「最大的圓盤」必定要最先從 A 移到 C。



然而如果想要移動這個目前位於 A 底部的最大的盤子，在它上方的 個圓盤都必須淨空，先移動到其他兩根棍子之一。

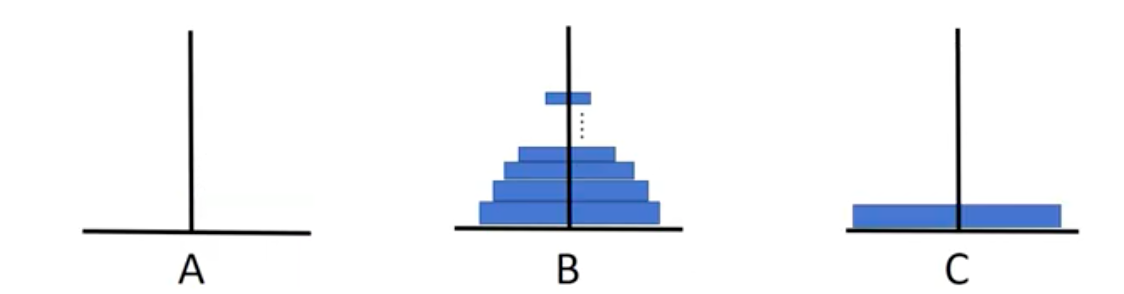


假設找到一個方法，把上面的 個盤子都移動到 B 了（這個方法會需要移動 次圓盤），那麼緊接著，就可以用「一次」移動把最大的盤子從 A 棍子移動到 C 棍子上。

經過上面的處理，目前的情況是

A 棍子：沒有盤子

B 棍子：除了最大盤子外的 個盤子C 棍子：最大的盤子



從目前的情況開始，繼續把 B 上的 個盤子移動到 C 棍子，就可以達成遊戲的目標。然而這同樣需要經過「把 個盤子移動到另一個棍子」的步驟，需要移動 次圓盤。



可以列出下面的等式：

把 個盤子從 A 移到 C =把 個盤子從 A 移到 B + 把最大的盤子從 A 移到 C + 把 個盤子從 B 移到 C

也就是說，

因此， 層河內塔的問題可以被拆分表示為「 層河內塔的問題」與「 層河內塔的問題」與另一個「 層河內塔的問題」。

（2）列出遞迴式

用遞迴式的形式表達：

當 時，直接把唯一的盤子移動過去即可，只需要一次移動； 時，前後各進行 ，中間還要進行一次 ，因此會得到 的「遞迴關係式」。

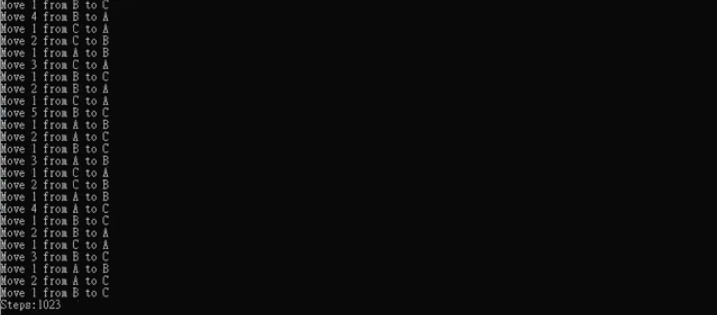
得到遞迴關係式後，就可以用之前介紹的各種方法來算出遞迴式的複雜度：

因此，

（3）輸出河內塔的過程

讓使用者輸入一整數 ，輸出搬動 層河內塔從 A 棍到 C 棍的所有「過程」

|  |  |
| --- | --- |
| 河內塔 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #include <iostream>using namespace std;// 把代表三根棍子的三個堆疊分為 from、to 和 othersint Hanoi (int N, char from, char to, char others){ // 邊界條件：只有一個盤子（最大的盤子）要搬動 if (N==1){ // 把這個盤子由 from 搬到 to  cout << "Move " << N << " from " << from << " to " << to << endl; return 1; }  // 一般情形 else { // 把 n-1 個盤子從 from 移動到 others int step\_1 = Hanoi(N-1, from, others, to); // 把 1 個盤子從 from 移動到 to  int step\_2 = 1; cout << "Move " << N << " from " << from << " to " << to << endl; // 把 n-1 個盤子從 others 移動到 to int step\_3 = Hanoi(N-1, others, to, from);  // 回傳需要移動的總次數  return step\_1 + step\_2 + step\_3;  } // end of if  } // end of Hanoi  int main(){ // 讓使用者輸入 N，代表要移動 N 層河內塔 int N; cout << "Please enter N:" << endl; cin >> N; int steps = Hanoi(N, 'A', 'C', 'B');  cout << "Steps:" << steps << endl;  return 0;  } |
| 執行結果 | |
| Please enter N:  >> 10  ...  Move 1 from B to C  Steps:1023 | |



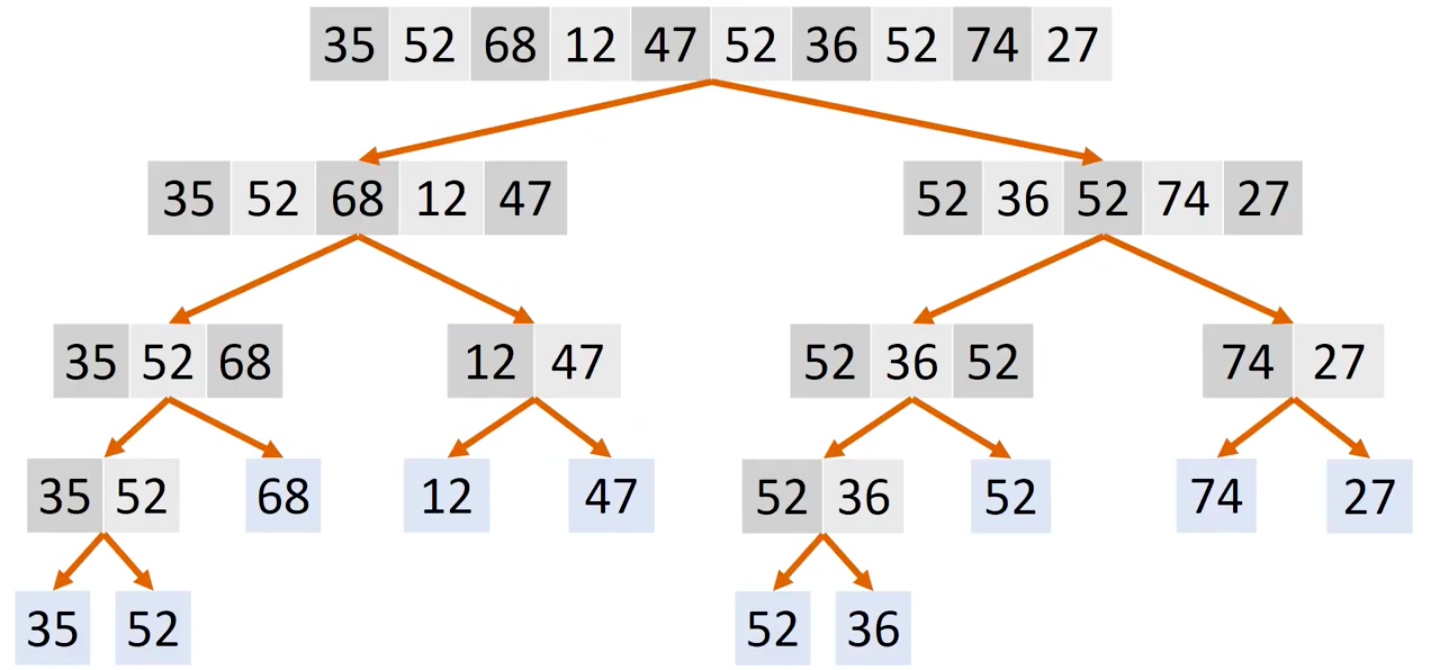
計算遞迴複雜度時，已經知道需要移動的次數是 ，輸出中 Steps 顯示步數為 ，符合先前的預想。

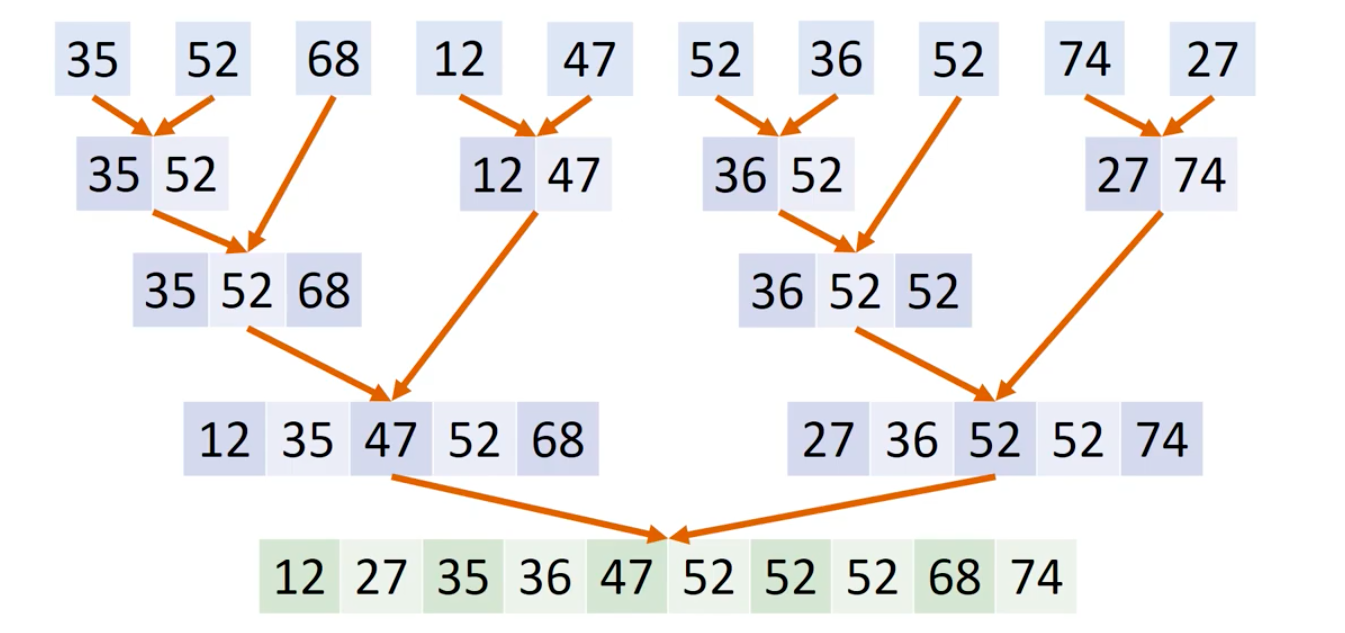
3. 合併排序與快速排序

「排序」的章節中介紹過的合併排序與快速排序其實也是分治法的應用。

（1）合併排序

合併排序是把資料切成兩組分別排序好，再把排序好的兩組資料融合在一起，它應用到的概念是「把兩個已經排序好的陣列融合在一起比較簡單」，所以可以一直把資料切割，切割成很多組資料後，分別排序完，最後再融合在一起。



從圖上來看，原本要排序的是一個長度為 10 的陣列，這個問題被切割成 10 個排序長度為 1 的子陣列的問題，之後，把針對每個子陣列得到的答案融合在一起，只要融合過程中能夠維持排序，就可以得到原問題的解答。

合併排序的程式碼如下：

|  |  |
| --- | --- |
| 合併排序 | |
| 1  2  3  4  5  6  7  8 | void Merge\_Sort(int data[], int start, int finish){  if (finish>start){  int middle = (finish+start)/2;  Merge\_Sort(data, start, middle);  Merge\_Sort(data, middle+1, finish);  Merge(data, start, finish, middle);  }  } |

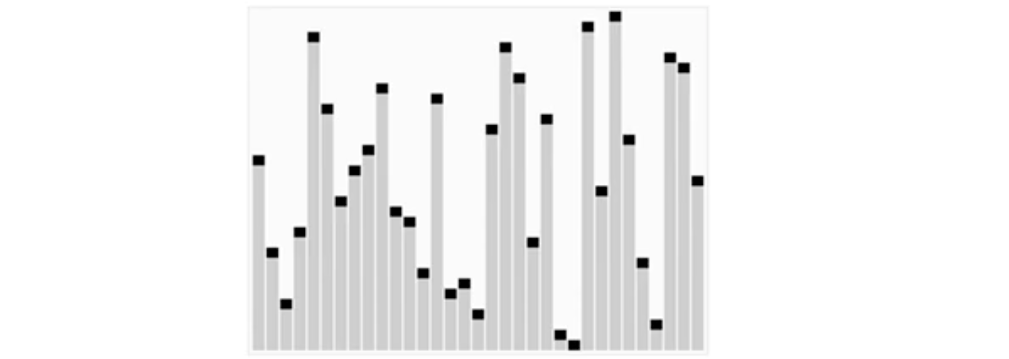
傳入一個陣列之後，陣列被分成左半部和右半部，兩個部分分別進行 Merge\_Sort（這就是 Divide & Conquer），之後，再以時間複雜度為 的過程融合 Merge 起來。

因為陣列長度 為 1 時不需要排序，複雜度是 ，而 時，排序問題被拆解成左邊排序和右邊排序，最後融合起來，因此可以得到下面的遞迴式：

化簡一下：

計算複雜度：

（2）快速排序法



再來是快速排序法，隨機選出一筆資料當基準點後，會把比它小的資料放在左邊、比它大的資料放在右邊。

|  |  |
| --- | --- |
| 快速排序法 | |
| 1  2  3  4  5  6  7 | void Quick\_Sort(int data[], int start, int finish){  if (start < finish){  int pivot = Partition(data, start, finish);  Quick\_Sort(data, start, pivot - 1);  Quick\_Sort(data, pivot + 1, finish);  }  } |

從程式碼來看，傳入一個陣列後，會首先進行一次複雜度為 的 Partition，之後再以 pivot 為基準，把問題分成「排序 pivot 的左邊」與「排序 pivot 的右邊」（Divide and Conquer），每個小問題都解決後，就可以得到原問題的答案。

假設每次選取 pivot 可以剛好把資料分成一半，那麼快速排序的複雜度和合併排序的複雜度一致，遞迴式同樣是：

平均而言，計算出的複雜度一樣是 。

4. 最大子數列問題

最大子數列問題指的是：給定一個陣列，陣列中的值有正有負，請找出一區間[a,b] 使區間內的「元素總和」最大，並回傳該元素總和。



給定如上的陣列，如果選定 4 到 6 的區間，其中的元素和是 ，選取 6 到 -2 的區間，元素和則是 。就像這樣，目標是選出一個區間，使得算出來的總和最大。

（1）暴力解

如果使用暴力解，用一個二維陣列來儲存 的和，那麼要用兩層 for 迴圈分別指定區間的開始位置和結束位置，再來，還要用一層迴圈來加總區間內的元素，可以簡單看出暴力解的複雜度高達 。

|  |  |
| --- | --- |
| 暴力解求最大子數列 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class Solution{  public:  int maxSubArray(vector<int>& nums){  int n = nums.size();  int sum[n][n]; // Should use malloc or vector instead;  int maximum = -2147483648; // -int\_max  // 指定區間開始位置  for (int start=0 ; start<n ; start++){  // 指定區間結束位置  for (int finish=start ; finish<n ; finish++){  sum[start][finish] = 0;  // 計算區間的元素和  for (int k = start ; k<=finish ; k++){  sum[start][finish] += nums[k];  }  if (sum[start][finish]>maximum){  maximum = sum[start][finish];  }  } // end of middle for  } // end of outer for  return maximum;  } // end of maxSubArray  }; // end of Solution |

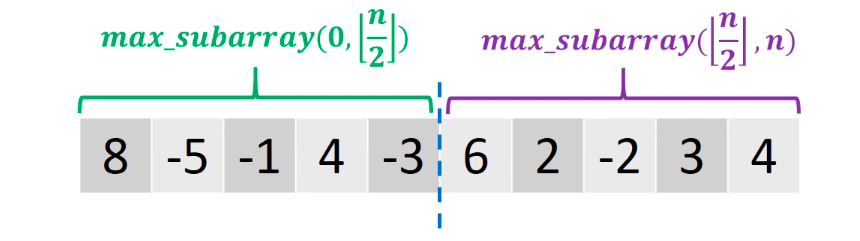
（2）暴力解的優化

一種優化的想法是把 [, ] 這個區間的和轉換成 這個區間的和與 區間和的差（比如 ）。在開始計算各個區間之前，先計算每一個從 0 開始的區間和：

|  |  |
| --- | --- |
| 暴力解求最大子數列（優化版） | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | class Solution{  public:  int maxSubArray(vector<int>& nums){  int n = nums.size();  // 儲存每個從開頭開始的區間的和  int sum\_1\_to\_n[n];  // 儲存所有區間的和  in sum[n][n];  int maximum = -2147483648; // -int\_max  // 先計算每個從開頭開始的區間的元素和，複雜度  for (int i=0 ; i<n ; i++){  if (i==0)  sum\_1\_to\_n[i] = nums[0];  else  sum\_1\_to\_n[i] = sum\_1\_to\_n[i-1] + nums[i];  }  // 利用 [0, finish] 和 [0, start-1] 兩個區間和的差  // 計算其他區間的元素和，複雜度降到  for (int start=0 ; start<n ; start++){  for (int finish-start ; finish<n ; finish++){  // start!=0  if (start){  // 這層的複雜度從 優化成  sum[start][finish] = sum\_1\_to\_n[finish] -  sum\_1\_to\_n[start-1];  }  // start==0  else{  sum[start][finish] = sum\_1\_to\_n[finish]  }  if (sum[start][finish]>maximum){  maximum = sum[start][finish];  }  } // end of inner for  } // end of outer for  } // end of maxSubArray  }; // end of Solution |

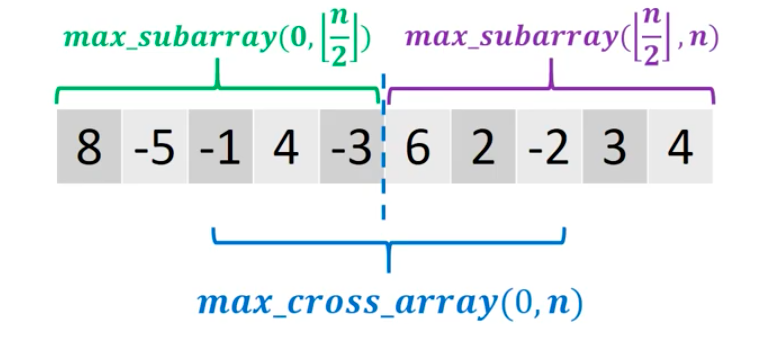
前面的一層迴圈複雜度是 ，後面的雙重迴圈複雜度是 ，總複雜度成功降為 。

（3）用分治法解最大子數列問題

試著拆解問題，下圖中，如果把陣列拆成左右兩邊，那麼最大值區間會出現在左半邊還是右半邊呢？（這樣分割對嗎？）

假設這樣分割是對的，那麼要找到整個數列中最大的區間，就先找數列左半邊中「和最大的區間」，再與數列右半邊中「和最大的區間」比較，兩者中較大者就是整個數列中最大的區間了：

但是很顯然，有一些區間不會完全在左半邊，也不會完全在右半邊，而是跨越了中間，這提醒了我們使用分治法時，要特別注意分割後的子問題是不是涵蓋了原問題的所有可能。



修正後的切割結果由三種可能組成：區間整個在左半邊、區間整個在右半邊，和區間跨越了中間，要回傳的是針對這三種可能各自求出最大和的區間後，有最大結果的那一個：

A.

B.

C.

前兩種可能性都與在整個數列中尋找最大和區間是作法相同，可以採用遞迴解決，但第三種可能性如何處理呢？

因為第三種可能性要求區間「跨越陣列中間」，所以陣列中間的那個元素一定被包含在這個 cross\_array 中，cross\_array 自己也有三種情形：

A. 只包含從這個中間元素出發，往左的若干個元素

B. 只包含從這個中間元素出發，往右的若干個元素

C. 從這個中間元素出發，既包含往左走的元素，也包含往右走的元素

不過這三種可能中，假設只往左走可以讓元素和變得更大（這個包含中間元素的往左區間和大於只有中間元素自己），只往右走也可以讓元素和變得更大，那麼區間同時往左也往右擴張，就會比單獨往左大，也比單獨往右大。

也就是說，一路往左擴張到底，看中間出現過的最大數字，作為「往左擴張」的最大值，再來，一路往右擴張到底，看中間出現過的最大數字，作為「往右擴張」的最大值，如果兩個數字都是正的，就把它們加起來，變成「往左擴張也往右擴張」可以得到的最大值（記得要包含中間元素自己的值）。

上面的例子中，最大的 元素和為中間元素 -3 加上 再加上 ，得到 值 16。

綜合來說，分治法的 Divide & Conquer 的精神體現在下面的程式碼裡，求最大子數列和的問題被分成三個子問題，三個子問題的解再融合出原問題的解：

|  |  |
| --- | --- |
| Divide & Conquer | |
| 1  2  3 | max\_left = maxSubArray(data\_left);  max\_right = maxSubArray(data\_right);  max\_center = maxCrossArray(data); |

|  |  |
| --- | --- |
| Combine | |
| 1  2  3  4  5  6 | if (max\_left>=max\_center && max\_left>=max\_right)  return max\_left;  else if (max\_right>=max\_center && max\_right>=max\_left)  return max\_right;  else  return max\_center; |

（4）分治法找尋最大子數列的時間複雜度

因為要取得 max\_left 和 max\_right 的值都需要 ， 是整個原數列要花費的時間，而要取得 max\_center 則會花費 時間（一路往左加到底和往右加到底），所以總複雜度與合併排序和快速排序相同，為 。

到了動態規劃的章節，還可以進一步把時間複雜度降到 。

（5）實作 max\_cross\_array

在實作「尋找最大子數列」的整個程式前，先來看尋找 max\_cross\_array 的程式碼：

|  |  |
| --- | --- |
| max\_cross\_array | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | int maxCrossArray(vector<int>& nums){  // 取得中間元素的值  int max\_center = nums[(len-1)/2];  // 宣告並初始化其他變數  int index\_left = -1, index\_right = 1, left\_sum = 0, right\_sum = 0, max = 0;  // 在超出左邊邊界前，取得往左途中加總出的每個和中最大者  while((len-1)/2+index\_left>=0){  left\_sum += nums[(len-1)/2+index\_left];  if (left\_sum>max)  max = left\_sum;  index\_left--;  }  // max\_center 加上往左擴張可以變大的值  // max 一開始被初始化成 0，所以不會在迴圈中變成負值  max\_center += max;  // 往右擴張  while((len-1)/2+index\_right<len){  right\_sum += nums[(len-1)/2+index\_right];  if (right\_sum>max)  max = right\_sum;  index\_right++;  }  // 再加上往右擴張可以變大的值  max\_center += max;  return max\_center;  } |

（6）LeetCode#53. 最大子數列 Maximum Subarray

A. 題目：

給定一個整數陣列 nums，找到其中的連續子陣列，其元素總和為最大，並回傳該元素總和。

B. 題目來源：

https://leetcode.com/problems/maximum-subarray/

C. 輸入與輸出範例：

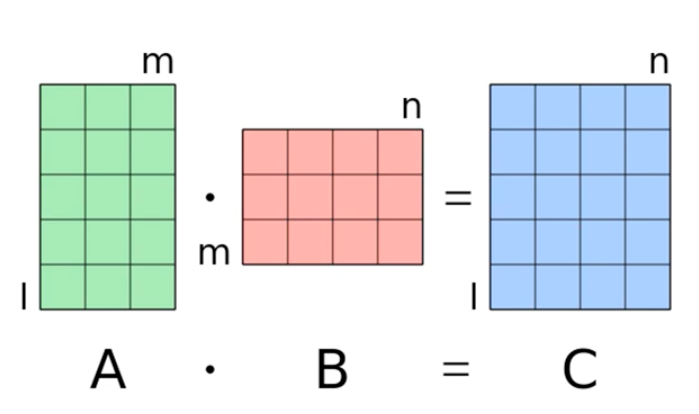
輸入：nums = [-2,1,-3,4,-1,2,1,-5,4]

輸出：Output：6

輸出的 6 是 [4,-1,2,1] 這個子數列的元素和。

|  |  |
| --- | --- |
| 最大子數列 Maximum Subarray | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | class Solution{  public:  int maxSubArray(vector<int>& nums){  // 取得整數陣列 nums 的長度  int len = nums.size();  // 例外處理：陣列中沒有資料，回傳整數最小值  if (len==0)  return -2147483648;  // 例外處理：陣列中只有一筆資料，回傳該筆資料  if (len==1)  return nums[0];  // 一般情況  // 訂出 data\_left 和 data\_right 的範圍  vector<int> data\_left(nums.begin(), nums.begin() + (len-1)/2);  vector<int> data\_right(nums.begin() + (len-1)/2, nums.end());  // 分成子問題處理  int max\_left = maxSubArray(data\_left);  int max\_right = maxSubArray(data\_right);  int max\_center = maxCrossArray(nums);  if (max\_left>=max\_center && max\_left>=max\_right)  return max\_left;  else if (max\_right>=max\_center && max\_right>=max\_left)  return max\_right;  else  return max\_center;  } // end of maxSubArray  int maxCrossArray(vector<int>& nums){  int len = nums.size();  int max\_center = nums[(len-1)/2];  int index\_left = -1, index\_right = 1, left\_sum = 0, right\_sum = 0;  int max = 0;  while((len-1)/2+index\_left>=0){  left\_sum += nums[(len-1)/2+index\_left];  if (left\_sum>max)  max = left\_sum;  index\_left--;  }  max\_center += max;  while((len-1)/2+index\_right<len){  right\_sum += nums[(len-1)/2+index\_right];  if(right\_sum>max)  max = right\_sum;  index\_right++;  }  max\_center += max;  return max\_center;  } // end of maxCrossArray  }; // end of Solution |

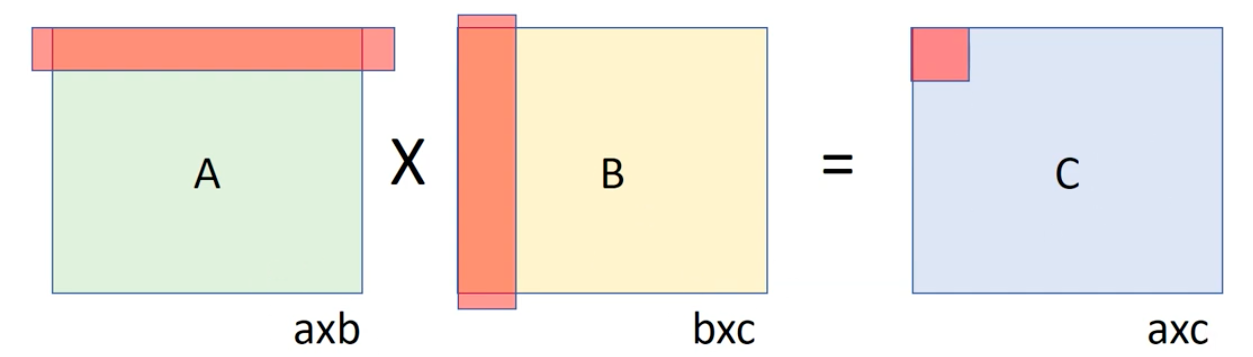
5. 矩陣相乘

再來，電腦科學中常常進行的「矩陣相乘」也隱含了分治法的觀念。

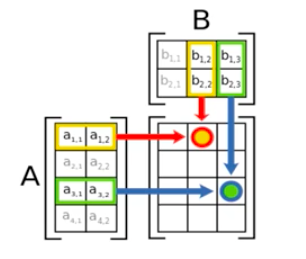
一般來說，一個大小為 的矩陣 A 乘上一個大小為 的矩陣 B，會得到一個大小為 的矩陣 C。在本節中，只考慮兩個大小同為 的正方形矩陣相乘的情形。

類似這樣的工作在機器學習、機械手臂的控制等工程中都很常用到，因此如果能夠化簡矩陣相乘的算法複雜度，就能大幅增加各種相關工程的執行速度。

（1）矩陣相乘的複雜度

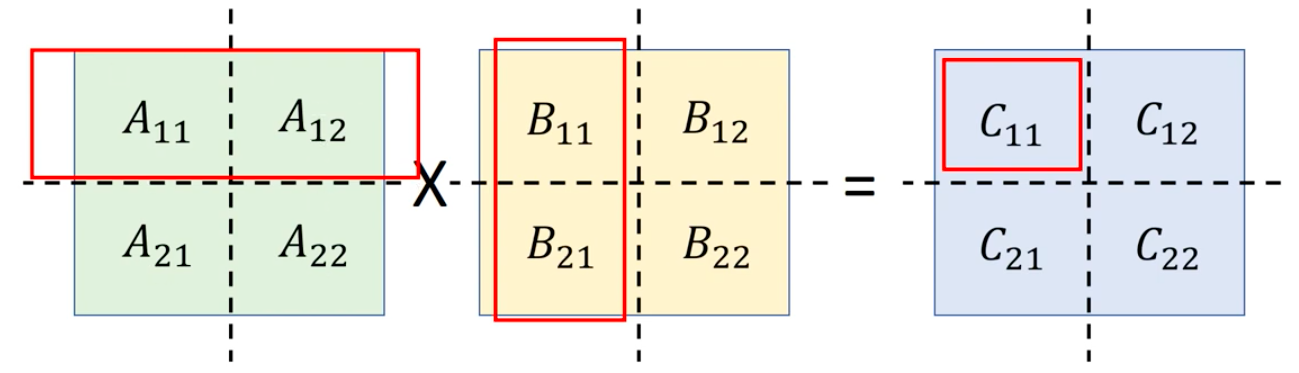


根據學校裡教的矩陣相乘規則，C 矩陣中的元素可以由下列式子求得：

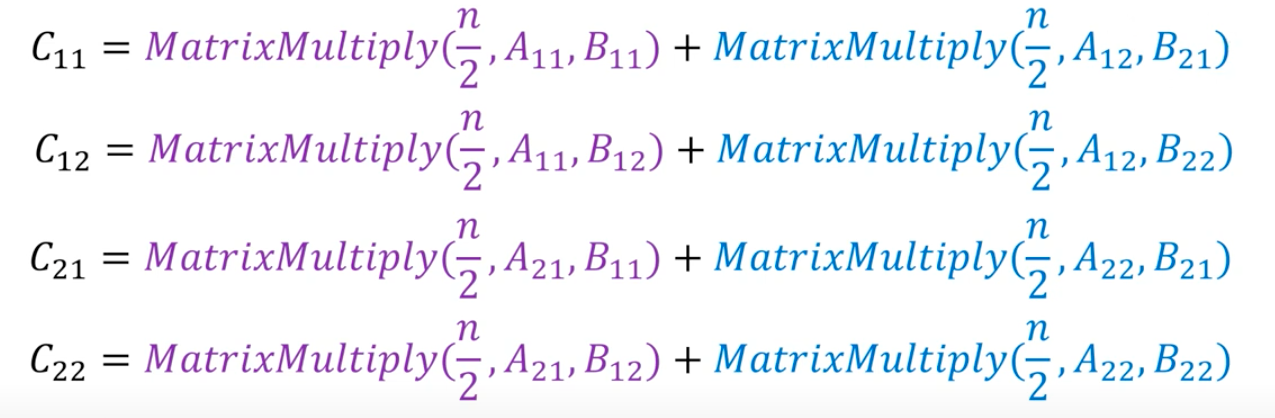
若為 的矩陣相乘時，要求得一個 C 矩陣中的元素，需要進行 次相乘後，再把答案相加才能得到。

而 C 矩陣本身的大小也是 ，共有 個元素。計算每個元素花費 的時間，所以計算整個 C 矩陣的複雜度為 。

（2）使用分治法進行子矩陣相乘



根據線性代數課程中教過的內容，矩陣相乘也可以使用分治法：把 A 矩陣切割成 4 個子矩陣 ，B 矩陣也切割成 4 個子矩陣 ，對 C 矩陣同樣進行切割，可以列出下面的關係式：



但是切割之後，雖然變成了大小 的陣列之間相乘，但是總共要進行 8 次這樣的相乘，最後還要把結果加起來（總共加 C 中的 個元素次），各個步驟的複雜度如下：

列出遞迴式：

可以計算出 ，並沒有比較好。

究竟有沒有辦法改善呢？

（3）Strassen algorithm

Strassen algorithm 關注的是降低上面遞迴式中遞迴的呼叫次數，透過這個算法，可以把呼叫次數由 8 次減少到 7 次。

它的概念有點像要得到 這個式子結果，直接計算要進行 4 次乘法、3 次加法，但是化為 後，只要進行兩次加法和一次乘法即可。

具體而言，已經有下面的幾個式子：

Strassen 算法中進一步引入 7 個新的矩陣：

透過這 7 個矩陣間的相加減，可以得到 C 矩陣的各部分：

Strassen 算法的遞迴關係如下：

|  |  |
| --- | --- |
| Strassen Algorithm | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | // A、B 矩陣大小都只有 1x1 時                              } |

就像這樣，遞迴的呼叫次數減少，遞迴式變成：

其中 來自最後加總得到 C 矩陣各部分的過程， 則會在提到支配理論時介紹如何計算出來。

透過減少遞迴的呼叫次數，運算次數從暴力解的 （A 是某個係數）降為 Strassen 算法的 （B 是另一個係數）。

因為 Strassen 算法中加減乘除比較多，使得 [係數B] > [係數A]，所以只有在矩陣較大（n 較大）的時候，才會比暴力解快，另外，過程中容易有溢位或浮點數運算誤差的問題，也會佔用多餘空間（要儲存 矩陣）。即使如此，機器學習時常常運算的是大小在數千乘數千以上的矩陣相乘，因此 Strassen 算法就顯得很實用。

6. 選擇問題

再來是選擇問題，選擇問題指的是：

A. 任意傳入一陣列

B. 指定常數 ， 陣列長度

C. 回傳陣列中第 大的數字



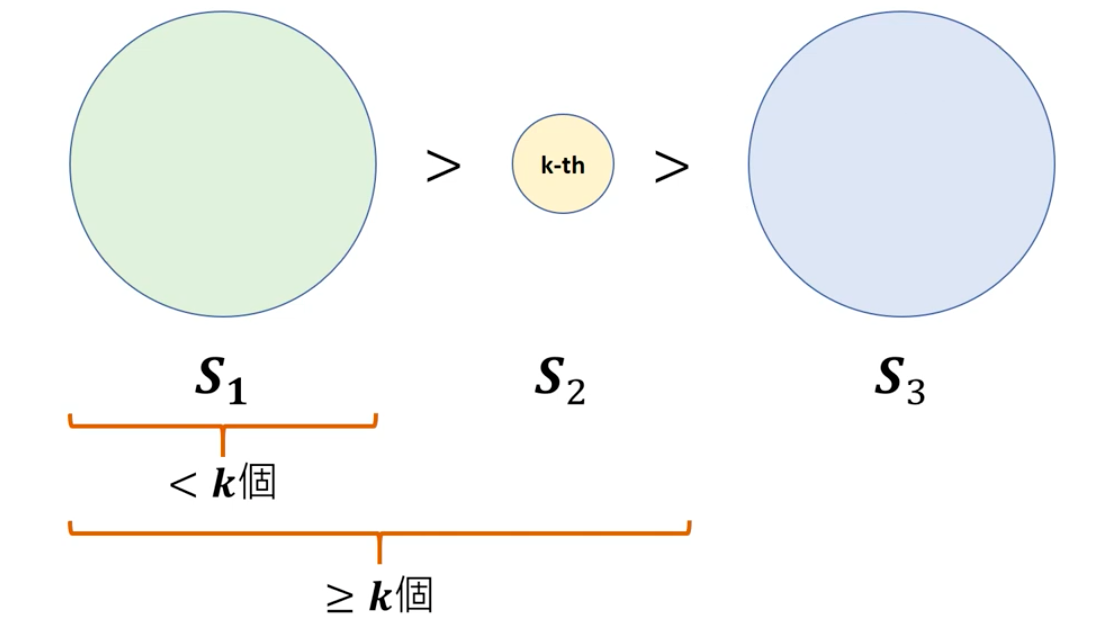
比如給定如上的陣列，給定 時，回傳陣列中最大的 74， 時，回傳第二大的 68，其餘以此類推。

要如何解決選擇問題呢？第一個想法是先對陣列做排序，排序完後就可以直接選取出第 大的數字，這樣一來選擇問題就被歸約為排序問題，也保證了一定可以在 的時間內解決，但是有沒有更好的算法呢？

（1）Prune and Search

先介紹分治法中的一個精神：Prune and Search。給定一個集合

，

目標是找出其中第 大的元素。

A. 首先，挑出一個元素 a（pivot）

B. 利用 a，把集合 S 區分為三個區塊

a. 大於 a 的集合：

b. 等於 a 的集合：

c. 小於 a 的集合：

C. 再來，分成三種狀況（： 中的元素數量）

a. ：，那麼第 大的元素一定在 中

b. ：

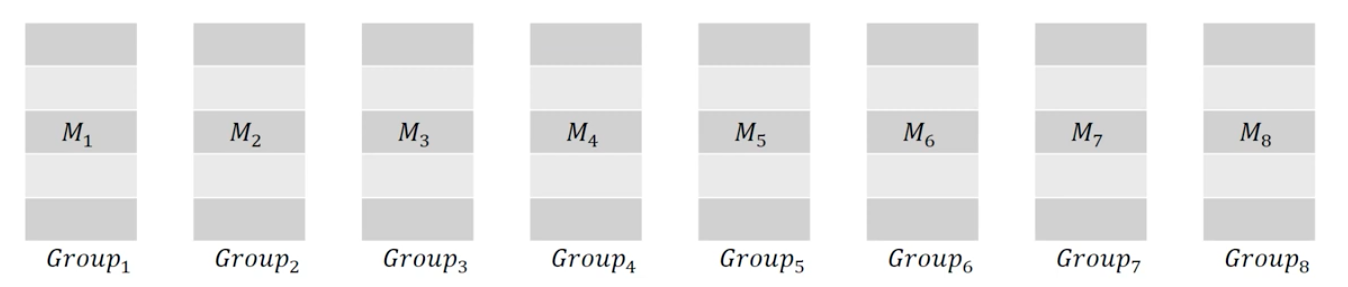
，那麼目標在 中， 中每個元素都等於 a，

因此 a 就是答案

c. ：目標在 中

在挑選 pivot 時，理想的狀況是挑選一次後，一定可以刪減掉固定比例的資料，才不會出現要刪非常多次的情形。既然會分成 3 個集合，那就希望資料可以平均分散成三等份，只是挑選 pivot 的算法本身一定不能太複雜，否則光是挑選的過程就會拖累整個算法的複雜度，這樣就沒有幫助了。

（2）Median of Median （MoM）

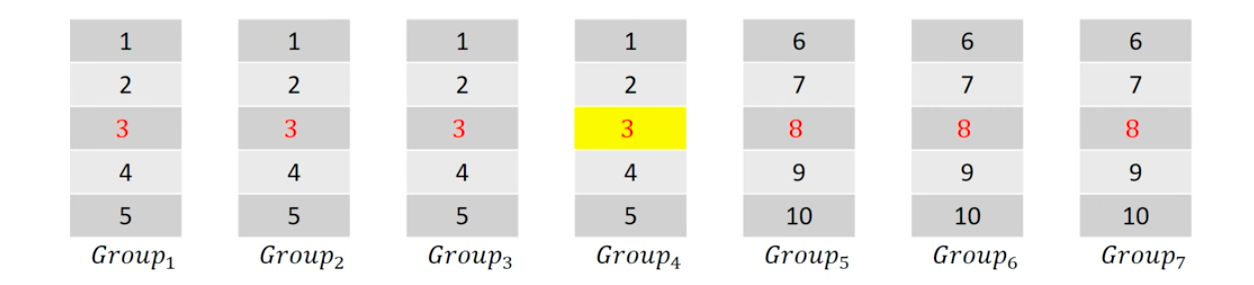


常見挑選 pivot 的算法是找出中位數的中位數：

A. 把資料分成五個五個一組

B. 分別取出每組（五個元素）的中位數

C. 再來，取出「所有中位數組成的集合」的中位數，也就是 MoM



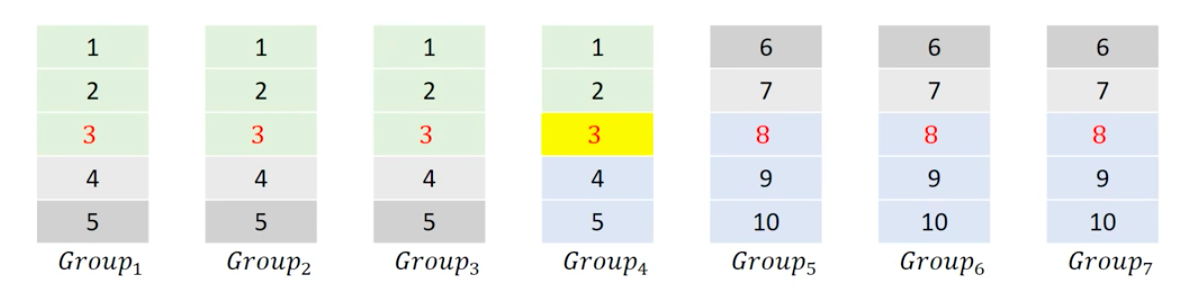
但是，中位數的中位數 MoM 並「不」一定是所有資料的中位數。像上圖中，每五筆資料分成一組後，總共分出了七組，這七組各自的中位數再取出中位數，得到第四組的 3，然而 3 並不是所有數字的中位數，因為簡單觀察一下，比 3 小的數字只有中央的 3 的「左上方」的數字，其他地方的數字都比 3 來得大，因此 3 明顯不是全部資料的中位數，事實上，全部共 35 個數字的中位數是 5。

為什麼不直接取出所有資料的中位數呢？因為並不存在一個算法可以很快的（比取得 MoM 更快）得到所有資料真正的中位數，因此才以 MoM 來代替。

MoM 仍有其價值，因為：

A. MoM 至少會 的資料

B. MoM 至少會 的資料



延續對上圖的觀察，中間的 3 至少會大於等於左邊一半資料的 （），也就是每個 1、2、3、4、5 中的 1、2、3 這三個橫列；另外，中間的 3 至少也會小於右邊一半資料的 ，因為它必定小於等於右邊一半資料的下面三個橫排。因為 ，所以發現 MoM 有「至少 的資料」，與「至少 的資料」兩個性質。

知道 MoM 可以用來切割資料後，就得到了一個較有效率的 pivot 挑選方法：

A. 將資料集合 S 分成（）筆資料，每組有 5 筆，最右邊不足 5 個的

組用「無限大」補足：

B. 排序每一組組內的五筆資料：，因為所需運算次數不受 影響

C. 找出每個組別內的中位數：

D. 重複步驟 ，完成後取出所有資料的 MoM：

E. 把 S 分成 、、：

F. 分成三種狀況加以判斷：

a. ：目標在

b. ：目標在 ，a 就是答案

c. 目標在

在 F 分出的三種狀況間，至少可以排除 的資料，這樣一來，步驟 F 需要的總時間就是 。

列出下列遞迴式後（不適用 的情形），可以得到 ：

複雜度的估計留給讀者，可以選用數學式或遞迴樹來證明（提示：先化簡為）。總結而言，透過採用 MoM 來當作 pivot 的作法，順利將選擇問題的複雜度由「排序後再取」的 降低到 。

7. 支配理論

再來，要介紹支配理論 Master theorem。

利用支配理論，可以很方便地在寫出遞迴式後進一步估計複雜度。不同於「數學解法」、「代換法」和「遞迴樹法」，支配理論更像一個「公式解」。

就像一元二次方程式可以藉由公式解：

直接得到答案一樣，支配理論也提供了可以直接套用以得到複雜度的公式。

（1）支配理論的推導

分治法中，一個大的問題可以分成數個小的問題（Divide & Conquer），接下來，需要再結合這些小問題的答案（Combine），因此列出的遞迴式常型如：，其中 a 代表分成了 a 個小問題， 是每個小問題的大小， 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：

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

又因為：

代入上式，得到：

---（a）

---（b）

---（c）

（b）式之所以可以轉換為（c）式，是因為如下的對數運算：

（上式左右同取 ）

因為求複雜度時，只需考慮 的指數最大的那項，因此只關注（c）式中第一項 和最後一項 之間誰大誰小即可。

比較 和 ，有下列三種情形：

A.

=> 是 的「上界」

=> 比 大，選擇看 這項就好

B.

=> 是 的「下界」

=> 比 大，選擇看 這項就好

C.

=> 和 一樣大

上式結果可由等差級數公式得到，交給讀者證明。

（2）重訪 Strassen 演算法

還記得 Strassen 演算法的遞迴式可以表示為：

利用 Master Theorem，比較 與 的大小，發現應取 ，也因此複雜度可寫成：

（3）重訪合併排序

參考先前合併排序的介紹，遞迴式如下：

得到 ，與支配理論的 比較，可得 。

與 中 的次方數相同，適用支配理論的第三種狀況（兩者大小相同時）：

比起利用數學解法等計算，使用支配理論的公式，可以更快得出遞迴式的複雜度。

8. 實戰練習

（1）LeetCode#169 多數元素 Majority Element

A. 題目

給定一個長度為 的整數陣列 ，回傳「多數元素」。這裡多數元素指的是「出現多於 次的元素」。測資中的每個陣列中都存在一個多數元素。

B. 出處：

https://leetcode.com/problems/majority-element/

C. 輸入與範例輸出

輸入：nums = [3,2,3]

輸出：3

D. 解題邏輯

使用本章介紹的分治法，將陣列分成左半部與右半部，分別取出兩個子陣列中出現次數最多的數字。

如果兩個子陣列中出現次數最多的數字相同，該數字就是所求多數元素；否則，假設兩邊出現最多次的數字分別為 x 和 y，出現次數分別為 count\_x 和 count\_y，則比較 count\_x 和 count\_y 的大小，決定回傳 x 或者 y （選出現次數多的）。

|  |  |
| --- | --- |
| 多數元素 Majority Element | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | class Solution{  // 傳入整數陣列、下界的索引值與上界的索引值  // 回傳介於傳入下界與上界間的區間中的多數元素  int majorityElement\_range(vector<int>& nums, int lower, int upper){  // 邊界處理：傳入區間已經縮小到沒有值  if (lower==upper)  return nums[lower];  // 一般情形  // 分成左右兩個子陣列遞迴處理  int middle = lower + (upper-lower)/2;  int left\_major = majorityElement\_range(nums, lower, middle);  int right\_major = majorityElement\_range(nums, middle+1, upper);  // 比較左邊與右邊多數元素    // 相同時任意回傳一邊  if (left\_major==right\_major)  return right\_major;  // 不同時，回傳出現次數較多者  int left\_count = 0, right\_count = 0;  for (int i=lower ; i<middle ; i++){  if (nums[i]==left\_major)  left\_count++;  }  for (int i=middle+1 ; i<upper ; i++){  if (nums[i]==right\_major)  right\_count++;  }  return left\_count>right\_count ? left\_major : right\_major;  }  public:  int majorityElement(vector<int>& nums){  // 把整個 nums 陣列傳入函式處理  return majorityElement\_range(nums, 0, nums.size()-1);  }  }; |

上述方法的遞迴式為：

後項 出現在「判斷應採左子陣列的多數元素或採右子陣列的多數元素」的過程，利用 Master Theorem，可以得到複雜度為 。

（2）LeetCode#240 搜尋二維陣列 II Search a 2D Matrix II

A. 題目

試寫一個有效率的演算法來搜尋大小為 的整數二維陣列中的某個目標值，若該值存在在陣列中，回傳 true，否則回傳 false。

該陣列有下列性質：

a. 每個橫列中，整數由左到大遞增

b. 每個直行中，整數由上到下遞增

B. 出處：

https://leetcode.com/problems/search-a-2d-matrix-ii/

C. 範例輸入與輸出

輸入：matrix =

[

[1,4,7,11,15],

[2,5,8,12,19],

[3,6,9,16,22],

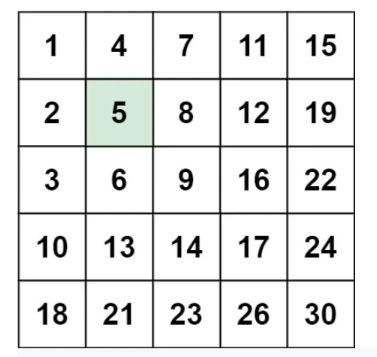
[10,13,14,17,24],

[18,21,23,26,30],

],

target = 5

輸出： true

D. 解題邏輯

選定一個出發點，如上面範例的右上角 15，如果要尋找的目標值比 15 小，則往左邊移動一格，因為只有左邊幾個直行才有比 15 小的值；目標值比 15 大時，則往下移動一格，因為只有下面幾個橫列才有比 15 小的值；如果前兩個情況都不符合，則目前選定的點就是目標值 15。

如果最後執行超出左方或下方邊界，就代表數列中並沒有要尋找的目標值，回傳 false。

這題是否有應用到分治法的概念呢？把原本的問題分成兩個子問題：當比目前所在這格的數字大時，子問題是在目前這格「下方」的子陣列中搜尋；比目前數字小時，子問題則是在目前這格「左方」的子陣列中搜尋。

|  |  |
| --- | --- |
| 搜尋二維陣列 II Search a 2D Matrix II | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | class Solution{  public:  bool searchMatrix(vector<vector<int>>& matrix, int target){  // 例外處理：陣列是空的  if (matrix.empty())  return false;  if (matrix[0].empty())  return false;  // 一般情形  // 取得陣列大小  int rows = matrix.size();  int cols = matrix[0].size();  // 從右上角的元素出發  int row\_now = 0;  int col\_now = cols-1;  // 超出左邊與下方邊界前繼續執行  while (row\_now<rows&&col\_now>=0){  // 找到目標，回傳 true  if (matrix[row\_now][col\_now]==target){  return true;  }  // 往左找  else if (matrix[row\_now][col\_now]>target){  col\_now--;  }  // 往下找  else {  row\_now++;  }  return false;  } // end of while  } // end of searchMatrix  }; // end of Solution |