19

搜尋與排序

With sobs and tears he sorted out
Those of the largest size …
—Lewis Carroll

Attempt the end, and never stand to doubt;
Nothing's so hard, but search will find it out.
—Robert Herrick

'Tis in my memory lock'd,
And you yourself shall keep the key of it.
—William Shakespeare

It is an immutable law in business that words are words, explanations are explanations, promises are promises — but only performance is reality.
—Harold S. Green

學習目標

在本章中，你將學到：

- 使用二分搜尋法在向量中搜尋已知的值。
- 使用 Big O 表示法來呈現以及比較搜尋和排序演算法的效率。
- 使用遞迴合併排序演算法排序一個向量。
- 了解常數時間、線性時間，以及平方時間演算法的特質。

本章綱要

19.1 簡介

搜尋 (searching) 資料是判斷資料中是否存在某個值 [此值稱作**搜尋鍵值 (search key)**]，若存在，就找出該值的位置。常見的搜尋演算法有兩種，一種是簡單的線性搜尋 (linear search) (已於第 7.7 節介紹)，另一種是速度較快，也較複雜的二分搜尋法 (binary search)，將於本章介紹。

排序 (sorting) 意指將資料依序擺放，通常是依照一或多個**排序鍵值 (sort key)** 做遞增或遞減排列。人名清單可依字母順序排序，銀行帳戶可依帳號排序，員工薪資記錄可依身分證字號排序等等。我們在前面學過插入排序法 (insertion sort) (第 7.8 節) 與選擇排序法 (selection sort) (第 8.6 節)。本章會介紹更有效率，也更複雜的合併排序法 (merge sort)。圖 19.1 整理出本書範例與習題所討論的搜尋與排序演算法。本章亦介紹 **Big O 表示法 (Big O notation)**，用來評估某個演算法的最差執行時間，也就是該演算法碰到最困難的情況時，可能要花多久時間解問題。

19.2 搜尋演算法

找電話號碼、存取網站與查字典都需搜尋大量資料。搜尋演算法的目標都一樣：找出符合已知搜尋鍵值的元素，如果該元素確實存在。但各演算法有諸多差異。主要差異在於搜尋所需耗費的成本。Big O 表示法便是表示成本的方式之一。對搜尋與排序演算法而言，此成本主要取決於資料元素的數量。

演算法	位置
搜尋演算法	
線性搜尋	第 7.7 節
二分搜尋	第 19.2.2 節
遞迴線性搜尋	習題 19.8
遞迴二分搜尋	習題 19.9
二元樹搜尋	第 20.7 節
鏈結串列的線性搜尋	習題 20.21
binary_search 標準函式庫函式	第 22.5.6 節
排序演算法	
插入排序法	第 7.8 節
選擇排序法	第 8.6 節
遞迴合併排序法	第 19.3.3 節
氣泡排序法	習題 19.5 與 19.6
貯桶排序法	習題 19.7
遞迴快速排序法	習題 19.10
二元樹排序法	第 20.7 節
sort 標準函式庫函式	第 22.5.6 節
堆積排序法	第 22.5.12 節

圖 19.1　　本書的搜尋和排序演算法

第 7 章討論過線性搜尋演算法，是最簡單、最易實作的搜尋演算法。我們現在用 Big O 表示法作為測量方式，討論線性搜尋法的效率。接著，我們介紹另一種更有效率，也更複雜、更難實作的搜尋演算法。

19.2.1　線性搜尋的效率

假設有個演算法只會檢查向量內的第一個元素是否等於第二個元素。若向量有 10 個元素，此演算法只要比一次。若向量有 1000 個元素，此演算法還是只比一次。實際上，此演算法跟向量的元素個數完全無關。此種演算法稱作**常數時間 (constant runtime)**， Big O 表示法寫成 **O(1**)。O(1) 的演算法並非眞的只比一次。O(1) 只是代表比較的次數是**常數**。它不會隨著向量大小而增長。若演算法只比較向量的第一個元素是否等於接下來的三個元素，那就要比三次，但 Big O 表示法還是寫成 O(1)。O(1) 通常稱作「階數爲 1」或簡稱「**1 階 (order 1)**」。

若演算法要比較向量的第一個元素是否等於向量中**任一個**元素，那最多要比 n － 1 次，n 代表向量的元素個數。若向量有 10 個元素，此演算法最多要比 9 次。若向量有 1000 個元素，則此演算法最多要比 999 次。n 越大，此運算式的 n 便會「主導」一切，那個減 1 就無足輕重了。Big O 是在 n 越來越大時表示主導的項目，而忽略不重要的項目。因此，一個總共要比 n－1 次的演算法 (如本段描述的) 就是 **O(n)**。O(n) 演算法稱作**線性時間 (linear runtime)**。O(n) 通常稱作「階數爲 n」或簡稱「**n 階 (order n)**」。

假設現在我們要測試整個向量中，是否有**任何**重複的元素。第一個元素必須跟所有其餘的元素比。第二個元素必須跟其餘的元素比，除了第一個元素以外 (因爲在第一次比過了)。第三個元素必須跟其餘的元素比，除了前兩個元素以外。最後，此演算法總共要比 (n － 1) + (n － 2) + ... + 2 + 1 次，也就是 $n^2/2 – n/2$ 次。只要 n 越大，n^2 便會主導此運算式，n 就變得不重要了。同樣的，Big O 會凸顯 n^2 項，而非 $n^2/2$。Big O 表示法常會省略分數部分，如後所述。

Big O 只關心當項目數量變多時，演算法時間的增長情形，而非處理的項目有多少。假設一個演算法要比 n^2 次。若有 4 個元素，就要比 16 次；有 8 個元素，就要比 64 次。在此演算法中，只要元素加倍，比較次數就加平方倍。假設另一個演算法要比 $n^2/2$ 次。若有 4 個元素，就要比 8 次；有 8 個元素，就要比 32 次。同樣的，只要元素加倍，比較次數就加平方倍。這兩個演算法都隨 n 平方成長，所以 Big O 忽略分數部分，把兩個都寫成 **O(n²)**，稱作**平方時間 (quadratic runtime)**，唸做「階數爲 n 平方」或簡稱「**n 平方階 (order n-squared)**」。

當 n 很小時，$O(n^2)$ 演算法對效能沒有什麼影響 (對今日每秒可執行十億個操作的個人電腦而言)。但 n 越來越大時，便會感受到效能下降了。若向量有 100 萬個元素，$O(n^2)$ 演算法需要一兆次操作 (每個操作可能要執行數個機器指令)。可能得花幾小時。若向量有十億個元素，就要一百萬兆次操作！可能要花上幾十年呢！$O(n^2)$ 演算法很容易寫，如前面章節所述。本章會介紹 Big O 更理想的演算法。這些高效能演算法比較不好寫，但因爲效率高，所以多花點心力也值得，尤其在 n 很大，且演算法要用在大型程式時。

線性搜尋演算法要花 O(n) 時間。最差情況下，每個元素都要檢查過，以判斷本向量內是否有搜尋鍵值。若向量大小加倍，演算法比較次數也得加倍。若搜尋鍵值位在向量的第一個元素，或靠得很前面，那線性搜尋當然超快。但我們要的演算法，是在平均情況下都跑的很快，即使搜尋鍵值在向量最後面也沒差。

線性搜尋是最容易實作的搜尋演算法，但跟其它搜尋演算法比起來就很慢。若程式要在很大的向量上執行多次搜尋，最好採用更有效率的演算法，如下一節介紹的二分搜尋法。

增進效能的小技巧 19.1

有時候最簡單的演算法效能會很差。但好處是容易撰寫、測試和除錯。有時得用複雜的演算法，以求最佳效能。

19.2.2　二分搜尋

二分搜尋演算法 (binary search algorithm) 的效能比線性搜尋高，但向量要先排序過。若向量排序後會搜尋很多次，或程式非常在乎效能，那麼用此演算法才值得。此演算法的第一次循環會測試向量中間的元素。若它就是搜尋鍵值，演算法便結束。假設向量從小到大排列，若搜尋鍵值比中間元素小，那麼此搜尋鍵值就不可能出現在後半段，因此演算法只繼續在前半段找 (也就是第一個元素到中間的前一個元素)。若搜尋鍵值比中間元素大，那麼此搜尋鍵值就不可能出現在前半段，因此演算法只繼續在後半段找 (也就是中間的後一個元素到最後一個元素)。每次循環會測試其餘部分的中間元素。若該元素也不等於搜尋鍵值，演算法就再砍掉一半元素。最後，演算法會找出那個等於搜尋鍵值的元素，或把子向量大小砍成零。

例如，以下的已排序向量有 15 個元素，

```
2  3  5  10  27  30  34  51  56  65  77  81  82  93  99
```

搜尋鍵值為 65。實作二分搜尋演算法的程式，會先檢查 51 是不是搜尋鍵值 (因為 51 是向量中間的元素)。搜尋鍵值 (65) 大於 51，所以 51 與 51 之前的前半段都會切掉 (所有元素都小於 51)。接著，演算法檢查 81 (其餘後半段向量的中間元素) 是不是搜尋鍵值。搜尋鍵值 (65) 比 81 小，所以 81 與大於 81 的元素都被切掉。經過兩回測試，此演算法已將要檢查的元素減為 3 個 (56、65 和 77)。演算法接著檢查 65 (就是搜尋鍵值)，並回傳此向量索引 (9) 包含了 65。此例中，演算法只要比三次，就能找出是否有元素等於搜尋鍵值。用線性搜尋法就要比 10 次。[請注意：此例的向量有 15 個元素，所以總是找得到中間元素。若元素個數為偶數，中間數就在兩個元素之間了。本實作選擇較大的那個元素做中間數。]

圖 19.2-19.3 分別定義 BinarySearch 類別及其成員函式。BinarySearch 類別跟 LinearSearch (第 7.7 節) 很像，它也有個建構子、一個搜尋函式 (binarySearch)、一個 displayElements 函式、兩個 private 資料成員，以及一個 private 工具函式 (displaySubElements)。圖 19.3 的第 11–21 行定義了建構子。將此向量以亂數方式初始化爲 10-99 的 int 後 (第 17–18 行)，第 20 行在向量 data 上呼叫標準函式庫函式 sort。前面提過，二分搜尋演算法只能在排序過的向量上運作。函式 **sort** 需要兩個引數，用來指定要排序的元素範圍。程式以循環器 (將在 22 章詳細介紹) 來指定這兩個引數。vector 的成員函式 begin 和 end 傳回循環器，sort 函式利用這些循環器，指出要排序的範圍是整個 vector。

第24–54行定義了binarySearch函式。搜尋鍵值會傳入searchElement參數 (第 24 行)。第 26–28 行會計算目前搜尋部分的 low 索引、high 索引和 middle 索引。在函式一開始，low 是 0、high 是向量大小減 1、middle 是這兩個數的平均。第 29 行把代表找到位置的 location 初始化成－1，若找不到搜尋鍵值，就會傳回此值。31–51 行會重複執行，直到 low 比 high 大 (表示找不到元素) 或 location 不等於－1 時 (表示找到了搜尋鍵值) 才停止。第 43 行測試 middle 元素是否等於 searchElement。若是 true，第 44 行會將 middle 指派給 location。迴圈便終止，並傳回 location 給呼叫者。每次循環會測試一個值 (第 43 行)，並將向量其餘的值砍掉一半 (第 46 行或 48 行)。

```cpp
// Fig 19.2: BinarySearch.h
// Class that contains a vector of random integers and a function
// that uses binary search to find an integer.
#include <vector>
using namespace std;

class BinarySearch
{
public:
   BinarySearch( int ); // constructor initializes vector
   int binarySearch( int ) const; // perform a binary search on vector
   void displayElements() const; // display vector elements
private:
   int size; // vector size
   vector< int > data; // vector of ints
   void displaySubElements( int, int ) const; // display range of values
}; // end class BinarySearch
```

圖 19.2　BinarySearch 類別定義

```
// Fig 19.3: BinarySearch.cpp
// BinarySearch class member-function definition.
#include <iostream>
#include <cstdlib> // prototypes for functions srand and rand
#include <ctime> // prototype for function time
#include <algorithm> // prototype for sort function
#include "BinarySearch.h" // class BinarySearch definition
using namespace std;

// constructor initializes vector with random ints and sorts the vector
BinarySearch::BinarySearch( int vectorSize )
{
   size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
   srand( time( 0 ) ); // seed using current time

   // fill vector with random ints in range 10-99
   for ( int i = 0; i < size; i++ )
      data.push_back( 10 + rand() % 90 ); // 10-99

   std::sort( data.begin(), data.end() ); // sort the data
} // end BinarySearch constructor

// perform a binary search on the data
int BinarySearch::binarySearch( int searchElement ) const
{
   int low = 0; // low end of the search area
   int high = size - 1; // high end of the search area
   int middle = ( low + high + 1 ) / 2; // middle element
   int location = -1; // return value; -1 if not found

   do // loop to search for element
   {
      // print remaining elements of vector to be searched
      displaySubElements( low, high );

      // output spaces for alignment
      for ( int i = 0; i < middle; i++ )
        cout << "   ";

      cout << " * " << endl; // indicate current middle

      // if the element is found at the middle
      if ( searchElement == data[ middle ] )
         location = middle; // location is the current middle
      else if ( searchElement < data[ middle ] ) // middle is too high
         high = middle - 1; // eliminate the higher half
      else // middle element is too low
         low = middle + 1; // eliminate the lower half

      middle = ( low + high + 1 ) / 2; // recalculate the middle
   } while ( ( low <= high ) && ( location == -1 ) );

   return location; // return location of search key
```

圖 19.3 BinarySearch 類別成員函式定義

```
} // end function binarySearch

// display values in vector
void BinarySearch::displayElements() const
{
   displaySubElements( 0, size - 1 );
} // end function displayElements

// display certain values in vector
void BinarySearch::displaySubElements( int low, int high ) const
{
   for ( int i = 0; i < low; i++ ) // output spaces for alignment
      cout << "   ";

   for ( int i = low; i <= high; i++ ) // output elements left in vector
      cout << data[ i ] << " ";

   cout << endl;
} // end function displaySubElements
```

圖 19.3 BinarySearch 類別成員函式定義 (續)

圖 19.4 的第 22-38 行會重複執行，直到使用者輸入－1。對使用者輸入的每個數，此程式都會執行一次二分搜尋法，判斷它是否出現在向量中。此程式的第一行輸出是 int 向量內容，從小到大排列。當使用者要找 38 時，程式先測試中間元素 67 (以 * 表示)。搜尋鍵值比 67 小，所以程式切掉後半段，繼續測試前半段的中間元素。搜尋鍵值等於 38，所以程式傳回索引 3。

```
1  // Fig 19.4: Fig19_04.cpp
2  // BinarySearch test program.
3  #include <iostream>
4  #include "BinarySearch.h" // class BinarySearch definition
5  using namespace std;
6
7  int main()
8  {
9     int searchInt; // search key
10    int position; // location of search key in vector
11
12    // create vector and output it
13    BinarySearch searchVector ( 15 );
14    searchVector.displayElements();
15
16    // get input from user
17    cout << "\nPlease enter an integer value (-1 to quit): ";
18    cin >> searchInt; // read an int from user
19    cout << endl;
20
21    // repeatedly input an integer; -1 terminates the program
```

圖 19.4 BinarySearch 測試程式

```
      while ( searchInt != -1 )
      {
         // use binary search to try to find integer
         position = searchVector.binarySearch( searchInt );

         // return value of -1 indicates integer was not found
         if ( position == -1 )
            cout << "The integer " << searchInt << " was not found.\n";
         else
            cout << "The integer " << searchInt
               << " was found in position " << position << ".\n";

         // get input from user
         cout << "\n\nPlease enter an integer value (-1 to quit): ";
         cin >> searchInt; // read an int from user
         cout << endl;
      } // end while
   } // end main
```

```
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
26 31 33 38 47 49 49
          *
The integer 38 was found in position 3.


Please enter an integer value (-1 to quit): 91

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
                        73 74 82 89 90 91 95
                                  *
                                    90 91 95
                                        *
The integer 91 was found in position 13.


Please enter an integer value (-1 to quit): 25

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                      *
26 31 33 38 47 49 49
          *
26 31 33
    *
26
 *
The integer 25 was not found.


Please enter an integer value (-1 to quit): -1
```

圖 19.4　`BinarySearch` 測試程式 (續)

二分搜尋的效率

在最差情況下，以二分搜尋法搜尋一個 1023 個元素的向量 (排序過)，也只要比 10 次。反覆將 1023 除以 2 (因爲每次比較後，就可以將向量砍一半) 並往下捨去 (因爲我們也會捨去中間元素)，結果就是 511、255、127、63、31、15、7、3、1 和 0。1023 ($2^{10}-1$) 只要除以 2 除十次，就能得到 0，表示最多只會檢查這麼多次。除以 2 就等於是在二分搜尋演算法中進行一次比較。因此，一個有 1,048,575 個 ($2^{20}-1$) 元素的向量最多只要比 20 次，就能找到鍵值，而超過十億個元素的向量，最多也只要比 30 次。比線性搜尋快多了。若向量有十億個元素，線性搜尋平均要比五億次，但二分搜尋最多只要比 30 次！二分搜尋在已排序向量上的最大比較次數，便是看 2 的幾次方能超過向量元素個數，該次方數就是最大比較次數，表示成 $\log_2 n$。所有對數都差不多以相同速率增長，所以 Big O 中底數可以省略。所以二分搜尋法的 Big O 就是 **O(log n)**，稱作**對數時間 (logarithmic runtime)**，唸做「階數是 log n」或簡稱「**log n 階 (order log n)**」。

19.3 排序演算法

資料排序 (就是資料依某種特定順序來排序，例如由小至大或由大至小) 是最重要的計算應用之一。銀行會將所有的支票按帳戶號碼排列，所以在月底核對帳單時，便很容易找到每個帳戶的餘額。電話公司會將用戶按照姓氏排序，然後再按照名字排序，如此就很容易找到所需的電話號碼。實際上，所有組織都有資料需要排序，且資料量通常都不小。排序如此重要，造成大家競相研究。

排序的重點，就是不管用什麼演算法來排序，最終結果一定要一樣。不同的演算法只會影響執行時間和記憶體用量。前面章節介紹過選擇排序法和插入排序法，它們都很簡單，但效率很差。下一節會以 Big O 表示法探討這兩種演算法的效能。合併排序法是本章介紹的最後一個演算法，他的速度快很多，但較難實作。

19.3.1 選擇排序法的效率

選擇排序法很容易撰寫，但效率很差。此演算法的第一次循環會選出向量中的最小數，然後將它跟第一個元素交換。第二次循環選出第二小的數 (也就是其餘元素中最小的數)，並將它與第二個元素交換。持續執行下去，直到最後一次循環選出第二大的數，並將它跟倒數第二個元素交換，將最大元素留在最後。在第 i 次循環後，前 i 小的元素都以遞增順序放進向量的前 i 個元素裡了。

選擇排序法會循環 n－1 次，每次均選出剩餘元素中的最小元素，放進排序的位置上。而在第一次循環時，找出最小元素要 n－1 次的比較，第二次循環時要 n－2 次的比較，然後是 n－3、 ... 、3、2、1 次。總共要 n (n－1) / 2 或 $(n^2 - n)/2$ 次的比較。Big O 表示法會把小項目和常數忽略掉，留下最後的 Big O，就是 $O(n^2)$。

19.3.2　插入排序法的效率

插入排序法也是個簡單但速度慢的演算法。此演算法的第一次循環會挑出第二個元素，若它比第一個元素小，就把它跟第一個元素交換。第二次循環會挑出第三個元素，並把它跟前兩個元素比，然後插進適當位置，所以前三個元素就排好了。在此演算法的第 i 次循環，原本向量中的前 i 個元素就排好了。

插入排序法共循環 n－1 次，每次都將一個元素插入之前排好的元素之間的適當位置。在每次循環中，若要決定元素該插入哪裡，可能要跟前面排好的每個元素都比過一次。最差情況下，要 n－1 次的比較。每個重複敘述要跑 O(n) 次。在決定 Big O 表示法時，巢狀敘述表示要乘上比較次數。對每一次外層迴圈而言，內層迴圈也跑了某個次數。此演算法中，外層迴圈共循環 O(n) 次，而外層迴圈的每次循環中，內層迴圈又循環 O(n) 次，所以 Big O 就是 O(n* n)，也就是 $O(n^2)$。

19.3.3　合併排序法 (遞迴實作)

合併排序法 (Merge sort) 是個很有效率的演算法，但觀念比選擇排序法和插入排序法複雜。合併排序法會把要排序的向量切成兩個同樣大小的子向量，將每個子向量排好，再把它們合併成一個較大的向量。若元素個數為奇數，此演算法就切出兩個子向量，一個向量比另一個多一個元素。

合併排序法會找每個向量的第一個元素，也就是每個向量中最小的元素，來進行合併。合併排序法會挑兩個之中最小的那個，並把它放到較大向量的第一個元素。若子向量中仍有元素，合併排序法就找該子向量的第二個元素 (現在它是其餘元素中最小的了)，並將它跟另一個子向量的第一個元素比較。合併排序法會持續執行，直到較大向量填滿為止。

本範例的合併排序法是以遞迴方式實作。基本狀況就是一個元素的向量。只有一個元素的向量當然是排好的，所以若向量只有一個元素，合併排序法便立刻傳回。若向量的元素有兩個以上，遞迴步驟會把它切成兩個同樣大小的子向量，以遞迴方式排序每個子向量，再把它們合併成一個更大、排序好的向量。[同樣的，若元素個數為奇數，一個子向量會比另一個子向量多一個元素。]

假設演算法已將較小向量合併成一個排好的向量 A：

```
4   10   34   56   77
```

和 B：

```
5   30   51   52   93
```

合併排序法會將這兩個向量組成一個較大、排序好的向量。A 的最小元素是 4 (位於 A 的索引 0 位置)。B 的最小元素是 5 (位於 B 的索引 0 位置)。為了決定較大陣列的第一個元素為何，演算法會比較 4 和 5。A 的值比較小，所以 4 變成合併向量的第一個元素。演算法繼續比較 10 (A 的第二個元素) 跟 5 (B 的第一個元素)。B 的值比較小，所以 5 變成較大向量的第二個元素。演算法繼續比較 10 跟 30，10 變成合併向量的第三個元素，依次類推。

圖 19.5 定義類別 `MergeSort`，而圖 19.6 的第 22–25 行定義 `sort` 函式。第 24 行呼叫 `sortSubVector` 函式，引數是 0 和 `size - 1`。此引數對應到待排序向量的開始和結束索引，所以 `sortSubVector` 會對整個向量進行操作。第 28–52 行定義函式 `sortSubVector`。第 31 行測試基本狀況。假如向量大小是 0，此向量就是排好的，所以函式立刻傳回。若向量大小大於或等於 1，此函式就將向量切成兩部分，遞迴呼叫函式 `sortSubVector` 以排序兩個子向量，然後合併它們。第 46 行對向量前半段遞迴呼叫函式 `sortSubVector`，而第 47 行對向量後半段遞迴呼叫函式 `sortSubVector`。當這兩個函式呼叫傳回時，這兩個半段的向量都排好了。第 50 行在這兩個半段的向量上呼叫函式 `merge` (第 55–99 行)，以將兩個排好的子向量合併成一個較大、排好的向量。

```cpp
// Fig 19.5: MergeSort.h
// Class that creates a vector filled with random integers.
// Provides a function to sort the vector with merge sort.
#include <vector>
using namespace std;

// MergeSort class definition
class MergeSort
{
public:
   MergeSort( int ); // constructor initializes vector
   void sort(); // sort vector using merge sort
   void displayElements() const; // display vector elements
private:
   int size; // vector size
```

圖 19.5　`MergeSort` 類別定義

```cpp
   vector< int > data; // vector of ints
   void sortSubVector( int, int ); // sort subvector
   void merge( int, int, int, int ); // merge two sorted vectors
   void displaySubVector( int, int ) const; // display subvector
}; // end class SelectionSort
```

圖 19.5　MergeSort 類別定義（續）

```cpp
// Fig 19.6: MergeSort.cpp
// Class MergeSort member-function definition.
#include <iostream>
#include <vector>
#include <cstdlib> // prototypes for functions srand and rand
#include <ctime> // prototype for function time
#include "MergeSort.h" // class MergeSort definition
using namespace std;

// constructor fill vector with random integers
MergeSort::MergeSort( int vectorSize )
{
   size = ( vectorSize > 0 ? vectorSize : 10 ); // validate vectorSize
   srand( time( 0 ) ); // seed random number generator using current time

   // fill vector with random ints in range 10-99
   for ( int i = 0; i < size; i++ )
      data.push_back( 10 + rand() % 90 );
} // end MergeSort constructor

// split vector, sort subvectors and merge subvectors into sorted vector
void MergeSort::sort()
{
   sortSubVector( 0, size - 1 ); // recursively sort entire vector
} // end function sort

// recursive function to sort subvectors
void MergeSort::sortSubVector( int low, int high )
{
   // test base case; size of vector equals 1
   if ( ( high - low ) >= 1 ) // if not base case
   {
      int middle1 = ( low + high ) / 2; // calculate middle of vector
      int middle2 = middle1 + 1; // calculate next element over

      // output split step
      cout << "split:   ";
      displaySubVector( low, high );
      cout << endl << "         ";
      displaySubVector( low, middle1 );
      cout << endl << "         ";
      displaySubVector( middle2, high );
      cout << endl << endl;
```

圖 19.6　MergeSort 類別成員函式定義

```cpp

      // split vector in half; sort each half (recursive calls)
      sortSubVector( low, middle1 ); // first half of vector
      sortSubVector( middle2, high ); // second half of vector

      // merge two sorted vectors after split calls return
      merge( low, middle1, middle2, high );
   } // end if
} // end function sortSubVector

// merge two sorted subvectors into one sorted subvector
void MergeSort::merge( int left, int middle1, int middle2, int right )
{
   int leftIndex = left; // index into left subvector
   int rightIndex = middle2; // index into right subvector
   int combinedIndex = left; // index into temporary working vector
   vector< int > combined( size ); // working vector

   // output two subvectors before merging
   cout << "merge:   ";
   displaySubVector( left, middle1 );
   cout << endl << "         ";
   displaySubVector( middle2, right );
   cout << endl;

   // merge vectors until reaching end of either
   while ( leftIndex <= middle1 && rightIndex <= right )
   {
      // place smaller of two current elements into result
      // and move to next space in vector
      if ( data[ leftIndex ] <= data[ rightIndex ] )
         combined[ combinedIndex++ ] = data[ leftIndex++ ];
      else
         combined[ combinedIndex++ ] = data[ rightIndex++ ];
   } // end while

   if ( leftIndex == middle2 ) // if at end of left vector
   {
      while ( rightIndex <= right ) // copy in rest of right vector
         combined[ combinedIndex++ ] = data[ rightIndex++ ];
   } // end if
   else // at end of right vector
   {
      while ( leftIndex <= middle1 ) // copy in rest of left vector
         combined[ combinedIndex++ ] = data[ leftIndex++ ];
   } // end else

   // copy values back into original vector
   for ( int i = left; i <= right; i++ )
      data[ i ] = combined[ i ];

   // output merged vector
```

圖 19.6　MergeSort 類別成員函式定義 (續 1)

```
96      cout << "           ";
97      displaySubVector( left, right );
98      cout << endl << endl;
99   } // end function merge
100
101  // display elements in vector
102  void MergeSort::displayElements() const
103  {
104     displaySubVector( 0, size - 1 );
105  } // end function displayElements
106
107  // display certain values in vector
108  void MergeSort::displaySubVector( int low, int high ) const
109  {
110     // output spaces for alignment
111     for ( int i = 0; i < low; i++ )
112        cout << "   ";
113
114     // output elements left in vector
115     for ( int i = low; i <= high; i++ )
116        cout << " " << data[ i ];
117  } // end function displaySubVector
```

圖 19.6　MergeSort 類別成員函式定義 (續 2)

函式 merge 的第 70–78 行會重複執行，直到程式碰到其中一個子向量的結束處爲止。第 74 行測試究竟哪一個向量的第一個元素比較小。若左邊向量的元素較小，第 75 行就把它放進合併向量的位置上。若右邊向量的元素較小，第 77 行就把它放進合併向量的位置上。while 迴圈跑完時 (第 78 行)，其中一個子向量已完全放進合併向量中了，但另一個子向量還有剩資料。第 80 行測試左邊向量是否已結束。若是，第 82–83 行就把右邊向量的元素填入合併向量中。若左邊向量尚未結束，那麼一定是右邊向量結束了，第 87–88 行便將左邊向量的元素填入合併向量中。最後，第 92–93 行將合併向量複製回原始向量中。圖 19.7 建立並使用 MergeSort 物件。此程式的輸出顯示合併排序法的切割與合併過程，展現此演算法每個步驟的排序情形。

合併排序法的效率

合併排序法的效率，比插入排序法和選擇排序法快得多 (雖然圖 19.7 看來頗複雜，很難相信它眞的比較快)。看看第一次 (非遞迴) 呼叫函式 sortSubVector (第 24 行) 。這會產生兩個遞迴呼叫 sortSubVector 函式，並切出兩個子向量，每個大小約等於原始向量的一半，以及一個 merge 函式呼叫。最差情況下，此 merge 函式呼叫需要 $n-1$ 次比較來填滿原始向量，也就是 O(n)。(前面提過，要從兩個子向量中比出較小的那個元

素)。這兩個 sortSubVector 函式呼叫又會造成四個遞迴呼叫 sortSubVector 函式，每個呼叫均帶有一個子向量，大小約是原始向量的 1/4，以及兩個 merge 函式呼叫。在最差情況下，這兩個 merge 函式呼叫分別要比 n/2－1 次，加起來共比 O(n) 次。繼續下去，每個 sortSubVector 呼叫都會另外產生兩個 sortSubVector 呼叫和一個 merge 呼叫，直到演算法將向量切成只有一個元素的子向量爲止。每一層都要比 O(n) 次，以合併子向量。而每一層會將向量大小減半，所以向量大小加兩倍，就要多一層。向量大小加四倍，就要多兩層。此模式便是對數形式，會有 $\log_2 n$ 層。所以整體效能是 **O(n log n)**。

```cpp
// Fig 19.7: Fig19_07.cpp
// MergeSort test program.
#include <iostream>
#include "MergeSort.h" // class MergeSort definition
using namespace std;

int main()
{
   // create object to perform merge sort
   MergeSort sortVector( 10 );

   cout << "Unsorted vector:" << endl;
   sortVector.displayElements(); // print unsorted vector
   cout << endl << endl;

   sortVector.sort(); // sort vector

   cout << "Sorted vector:" << endl;
   sortVector.displayElements(); // print sorted vector
   cout << endl;
} // end main
```

```
Unsorted vector:
 30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54
          30 47 22 67 79
                         18 60 78 26 54

split:    30 47 22 67 79
          30 47 22
                   67 79

split:    30 47 22
          30 47
                22

split:    30 47
          30
             47
```

圖 19.7　　MergeSort 測試程式

```
merge:    30
             47
          30 47

merge:    30 47
                22
          22 30 47

split:             67 79
                   67
                      79

merge:             67
                      79
                   67 79

merge:    22 30 47
                   67 79
          22 30 47 67 79

split:                   18 60 78 26 54
                         18 60 78
                                  26 54

split:                   18 60 78
                         18 60
                               78

split:                   18 60
                         18
                            60

merge:                   18
                            60
                         18 60

merge:                   18 60
                               78
                         18 60 78
split:                            26 54
                                  26
                                     54

merge:                            26
                                     54
                                  26 54

merge:                   18 60 78
                                  26 54
                         18 26 54 60 78

merge:    22 30 47 67 79
                         18 26 54 60 78
          18 22 26 30 47 54 60 67 78 79

Sorted vector:
 18 22 26 30 47 54 60 67 78 79
```

圖 19.7　　MergeSort 測試程式 (續)

圖 19.8 整理出本書的搜尋與排序演算法，並列出其 Big O。圖 19.9 列出本章討論的 Big O 種類，以及各種 n 的數值，讓讀者了解其增長程度。

演算法	位置	Big O
搜尋演算法		
線性搜尋	第 7.7 節	O(n)
二分搜尋	第 19.2.2 節	O(log n)
遞迴線性搜尋	習題 19.8	O(n)
遞迴二分搜尋	習題 19.9	O(log n)
排序演算法		
插入排序法	第 7.8 節	$O(n^2)$
選擇排序法	第 8.6 節	$O(n^2)$
合併排序法	第 19.3.3 節	O(n log n)
氣泡排序法	習題 19.5 與 19.6	$O(n^2)$
快速排序法	習題 19.10	最差情況下：$O(n^2)$ 平均情況下：O(n log n)

圖 19.8　　搜尋和排序演算法的 Big O 值

n	近似十進位值	O(log n)	O(n)	O(n log n)	$O(n^2)$
2^{10}	1000	10	2^{10}	$10 \cdot 2^{10}$	2^{20}
2^{20}	1,000,000	20	2^{20}	$20 \cdot 2^{20}$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$30 \cdot 2^{30}$	2^{60}

圖 19.9　　常見 Big O 表示法的比較次數

19.4　總結

本章討論了資料搜尋和排序。我們討論了二分搜尋演算法，這個演算法比線性演算法 (7.7 節) 更快但卻較爲複雜。二分搜尋法只能運用在已經排序好的陣列上，每比一次，就砍掉陣列一半元素。你也學到了合併排序法，比插入排序 (7.8 節) 和選擇排序 (8.6 節) 更有效率。我們也介紹了 Big O 表示法，它可以幫助你表達演算法的效率。Big O 演算法會估計一個演算法的最差執行時間。Big O 值可以用來比較演算法的優劣，讓你選擇最有效率的演算法。在下一章，你將會學到可在執行時期增大或縮小的動態資料結構。

摘要

19.1 簡介

- 搜尋資料意即判斷搜尋鍵值是否在此資料中，若有，就找出它的位置。
- 排序就是將資料依序排列。
- 說明演算法效率的方式之一，就是採用 Big O 表示法 (O)，它表示該演算法要花多少力氣，才能解決問題。

19.2 搜尋演算法

- 各種搜尋演算法的主要不同點，就是它們要花多少力氣才能找出結果。

19.2.1 線性搜尋的效率

- 對搜尋和排序演算法而言，Big O 說明當資料量增加時，演算法所耗的時間力氣是如何隨之增加的。
- O(1) 演算法稱作常數時間。這並不是說眞的只比一次。它只是代表比較次數不會隨著向量大小而增長。
- O(n) 演算法稱作線性時間。
- Big O 的目的，是在 n 越來越大時，凸顯出主導部分，而忽略不重要的項目。
- Big O 表示法著重在演算法執行時間的增加率，所以常數可以忽略。
- 線性搜尋演算法要花 O(n) 時間。
- 最差情況下，線性搜尋要把每個元素都檢查過，以判斷是否有搜尋鍵值。若搜尋鍵值在向量最後面，甚至根本不存在，就會發生最差狀況。

19.2.2 二分搜尋

- 二分搜尋演算法的效能比線性搜尋高，但向量要先排序過。它適用在向量已經被排序好，然後要進行許多次搜尋的狀況下。
- 二分搜尋法的第一次循環會測試向量中間的元素。若它就是搜尋鍵值，演算法便傳回它的位置。若搜尋鍵值小於中間元素，二分搜尋就繼續找向量前半段。若搜尋鍵值大於中間元素，二分搜尋就繼續找向量後半段。二分搜尋法的每次循環都會測試其餘部分的中間元素，若找不到元素，就將其餘元素砍掉一半。
- 二分搜尋的效率比線性搜尋高，因爲每比一次，就砍掉向量一半元素。
- 二分搜尋演算法要花 O(log n) 時間。
- 若向量大小加倍，二分搜尋法也只要多比一次。

19.3.1 選擇排序法的效率

- 選擇排序法很簡單，但效率很差。
- 選擇排序法的第一次循環會選出向量中最小的數，然後將它跟第一個元素交換。第二次循環選出第二小的數 (也就是其餘元素中最小的數)，並將它與第二個元素交換。演算法就這樣一直進行下去，直到最後一回合找出第二大的元素，將其與倒數第二個元素交換，讓最大的元素留在最後一個元素位置爲止。演算法第 i 回合結束之後，最小的 i 個元素會被依序放在前 i 個元素位置。

19.3.2 插入排序法的效率

- 選擇排序法要花 $O(n^2)$時間。
- 插入演算法的第一次循環會挑出第二個元素，若它比第一個元素小，就把它跟第一個元素交換。第二次循環會挑出第三個元素，並把它跟前兩個元素比，然後插進適當位置。在此演算法的第 i 次循環，原本向量中的前 i 個元素就排好了。只需要 n－1 次循環。
- 插入排序法要花 $O(n^2)$ 時間。

19.3.3 合併排序法 (遞迴實作)

- 合併排序法的效率比選擇排序法和插入排序法高，但實作也較複雜。
- 合併排序法會把要排序的向量切成兩個同樣大小的子向量，將每個子向量排好，再把子向量合併成一個較大的向量。
- 合併排序法最基本的情況就是一個元素的向量。只有一個元素的向量當然是排好的，所以若向量只有一個元素，合併排序法便立刻傳回。合併排序法的合併部分會把兩個排好的向量 (裡面可能只有一個元素) 合併成一個較大、排序過的向量。
- 合併排序法會找每個向量的第一個元素，也就是每個向量中最小的元素，來進行合併。合併排序法會挑兩個之中最小的那個，並把它放到較大向量的第一個元素。若子向量中仍有元素，合併排序法就找該子向量的第二個元素 (現在它是其餘元素中最小的了)，並將它跟另一個子向量的第一個元素比較。合併排序法會持續執行，直到較大向量填滿爲止。
- 在最差情況下，第一次呼叫合併排序法要 O(n) 次，才能填滿最後向量內的 n 個元素。
- 合併排序法的合併部分是在兩個子向量上執行，每個子向量的大小約是 n/2。要建立每個子向量則需 n/2－1 次比較，兩個子向量加起來就要比 O(n) 次。此模式持續下去，每一層的子向量數量會加倍，但每個子向量內的元素會減半。
- 跟二分搜尋法類似，此種減半行爲會產生 log n 層，每層要比 O(n) 次，所以綜合效能是 O(n log n)。

術語

Big O 表示法 (Big O notation)
二分搜尋法 (binary search algorithm)
常數時間 (constant runtime)
線性時間 (linear runtime)
對數時間 (logarithmic runtime)
合併排序法 (merge sort)
O(1)
O(log n)
O(n log n)
O(n)
O(n^2)
1 階 (order 1)
log n 階 (order log n)
n 階 (order n)
n 平方階 (order n-squared)
平方時間 (quadratic runtime)
搜尋鍵值 (search key)
搜尋 (searching)
選擇排序法 (selection sort)
排序鍵值 (sort key)
`sort` 標準函式庫函式 (sort standard library function)
排序 (sorting)

自我測驗

19.1 在以下敘述填空：

a) 選擇排序法對包含 128 個元素的向量所耗時間，約是對包含 32 個元素的向量所耗時間的_______倍。

b) 合併排序法的效率是_______。

19.2 在二分搜尋法與合併排序法中，造成其 Big O 中的對數部分的主因為何？

19.3 插入排序法的哪些地方優於合併排序法？合併排序法的哪些地方優於插入排序法？

19.4 本書說，在合併排序法將向量切成兩個子向量後，它就排序這兩個子向量且合併它們。為什麼有些人覺得「排序這兩個子向量」這句話很奇怪呢？

自我測驗解答

19.1 a) 16，因為資料增加四倍，O(n^2) 演算法就要增加 16 倍的時間。b) O(n log n)。

19.2 這兩個演算法都有「切一半」的動作，以某種方式將某些東西減少一半。二分搜尋法在每次比完後，就把要繼續找的向量切一半。每次呼叫合併排序法時，它就把向量切一半。

19.3 插入排序法比合併排序法容易了解，也較易實作。合併排序法的效率是 (O (n log n))，比插入排序法的 (O (n^2)) 高多了。

19.4 某種程度上，它沒有真的去排序這兩個子向量。它只是把原始向量切一半，直到出現一個元素的子向量為止，一個元素的子向量當然是排好的。然後它將這些只有一個元素的子向量合併成另一個較大的子向量，這些較大向量之後也會合併，持續下去，合併出原本這兩個子向量。

習題

[請注意：大部分習題都跟 7-8 章習題重複。我們重新列出這些習題，方便讀者學習本章的搜尋和排序法。]

19.5 (泡浮排序法) 實作泡浮排序法，這是另一種簡單但效率差的方法。它稱作「泡浮排序法」或「下沈排序法」，因爲較小的值最後會「浮上」向量表面 (就是較前面的元素)，就像氣泡浮上水面一般，而較大的值會下沈到向量底部 (末端)。本技術採用巢狀迴圈，對向量進行多個階段的操作。每個階段都比較相鄰的元素。如果有一對數值是按遞增順序排列 (或兩個數值是相等的)，就讓這些數值保持原來的順序。若這兩個數值是遞減排列，就交換它們在向量中的值。

第一階段比較前兩個元素，若有需要就交換。然後比較第二跟第三個元素。本階段最後就是比較最後兩個元素，若有需要就交換。一個階段完成後，最大的元素就會放在最後。兩個階段後，前兩大的元素都放在最後兩個位置上。解釋爲何泡浮排序法是 $O(n^2)$演算法。

19.6 (改良式泡浮排序法) 請做以下修改，改善習題 19.5 泡浮排序法的效能：

a) 在第一階段中，最大的數目一定會放到向量最後一個元素的位置，在第二階段後，這最大的兩個數字就排好了，依此類推。我們不要在每個階段都比 9 次 (對 10 個元素的向量而言)，請修改泡浮排序法，在第二階段比 8 次，第三階段比 7 次，依次類推。

b) 向量內的資料有可能已經排好，或快要排好了，所以幹嘛非要跑 9 個階段不可 (對 10 個元素的向量而言)？請修改泡浮排序法，在每階段比對之後，檢查是否執行了任何的交換操作。如果沒有任何交換，表示資料一定已經排序完畢，所以程式就應該終止。如果執行過交換，則至少要再進行一個階段的比較。

19.7 (貯桶排序法) 貯桶排序法 (bucket sort) 開始有個一維向量，裡面是要排序的正整數，以及另一個二維整數向量，每一列是 0 到 9，每一行是 0 到 n－1，n 是要排序的元素個數。二維向量的每列元素都稱爲一個貯桶。撰寫一個 `BucketSort` 類別，內含一個 `sort` 函式，運作方式如下：

a) 將一維向量內每個元素的值，按照每個值的個位數字 (最右邊的數字) 放入貯桶向量的同一列內。例如，97 就放到列 7，3 放到列 3，100 放到列 0。這稱爲「分配階段」(distribution pass)。

b) 利用迴圈，逐列將貯桶向量的每列元素複製回到原來的向量中。此過程稱爲「搜集階段」。在一維向量中的數值現在順序爲 100、3 和 97。

c) 對剩下的位數 (十位數、百位數、千位數等等)，重複以上步驟。

在第二階段 (十位數)，100 放到第 0 列，3 放到第 0 列 (因爲它沒有十位數)，97 放到第 9 列。經過搜集階段之後，一維陣列中數值的順序爲 100、3、97。在第三階段 (百位數)，100 放到第 1 列，3 放到第 0 列，97 放到第 0 列 (放在 3 後面)。經過最後一個搜集階段後，原始向量就排好了。

注意，二維貯桶向量的大小，是原始向量的 10 倍。這種排序的效率比泡浮排序法高，但需要較多記憶體空間 (泡沫排序法只需要增加一個元素的空間)。這在於空間和時間之間的取捨：這貯桶排序法使用的記憶體比泡浮排序法要多，但執行效率較高。這個版本的貯桶排序法在每個階段，要將所有資料複製回原本的陣列。另一種可能，就是建立第二個二維貯桶向量，然後在二個貯桶向量之間重複交換資料。

19.8 (遞迴線性搜尋) 修改習題 7.33，使用遞迴函式 `recursiveLinearSearch` 對向量執行線性搜尋。函式應接收一個搜尋鍵值和起始索引作爲引數。若找到搜尋鍵值，則傳回該元素的向量索引；否則傳回－1。每個遞迴函式的呼叫應檢查向量內的一個元素值。

19.9 (遞迴二分搜尋) 修改習題 19.3，使用遞迴函式 `recursiveBinarySearch` 對向量執行二分搜尋。函式應接收一個搜尋鍵值、起始索引、和結束索引作爲引數。若找到搜尋鍵值，就傳回該向量索引。若找不到搜尋鍵值，就傳回－1。

19.10 (快速排序法) 快速排序法 (quicksort) 是一種遞迴排序技巧，它對一維向量採用以下基本演算法：

a) 分割步驟：拿出未排序向量的第一個元素，並決定它在已排序向量中的最後位置 (也就是說，它左邊的所有元素都比它小，它右邊的所有元素都比它大。稍後將介紹如何進行此步驟)。我們現在已將一個元素放到正確位置，並有兩個未排序的子向量。

b) 遞迴步驟：在每個未排序的子向量上執行分割步驟。每次對子向量執行分割步驟時，就有另一個元素會放在排序過陣列的最終位置上，並再建立二個未排序的子向量。當子向量只有一個元素時，該元素就在它最終位置上了 (因爲只有一個元素的向量就是排好的)。

基本的演算法看來很簡單，但是要如何決定每個子陣列第一個元素的最終位置呢？例如，看以下的數值 (粗體字的元素就是分割元素，它會放在排序過向量的最終位置)

37　2　6　4　89　8　10　12　68　45

從向量最右邊開始，將每個元素跟 37 比，直到比 37 小的元素爲止，然後將此元素跟 37 交換。第一個比 37 小的元素是 12，因此 37 會和 12 交換。新向量如下

12　2　6　4　89　8　10　**37**　68　45

以斜體顯示的元素 12，指出它剛和 37 交換過。

接著從向量左邊開始，但這次從 12 後面開始算，將每個元素跟 37 比，直到碰到比 37 大的數爲止，然後將該數與 37 交換。第一個比 37 小的元素是 89，因此 37 會和 89 交換。新向量如下

12　2　6　4　**37**　8　10　*89*　68　45

從陣列的右邊開始，但是在元素 89 之前，將每個元素與 37 比較，直到發現比 37 小的元素爲止。然後將 37 和這個元素交換。第一個比 37 小的元素是 10，因此 37 會和 10 交換。新向量如下

12 2 6 4 *10* 8 **37** 89 68 45

接著從向量左邊開始，但這次從 10 後面開始算，將每個元素跟 37 比，直到碰到比 37 大的數爲止，然後將該數與 37 交換。再也沒有元素比 37 大，所以當我們將 37 與它本身進行比較時，就知道 37 已位在排序過向量的最終位置上。37 左邊的值都比它小，37 右邊的值都比它大。

分割向量之後，會產生二個未經排序的子向量。在前述向量套用此模式後，會產生兩個未排序的子向量。比 37 小的子向量內有 12、2、6、4、10 和 8。比 37 大的子向量有 89、68 和 45。接著以遞迴方式繼續排序，以分割原始向量的方式分割這兩個子向量。

請根據前述討論，撰寫一個遞迴函式 `quickSortHelper` 來排序一維整數向量。此函式應接收原始待排序向量的起始索引和結束索引作爲引數。