Algorithms (演算法)

南台科技大學 電子系　黎靖

一、概述

「演算法」的中文名稱出自周髀算經(寫成於西漢中期，西元前一百年左右)；演算法(algorithm)名稱來自於"al-Khwarizmi"，這是一個約在西元 780 年出生於伊拉克(Iraq)巴格達(Baghdad)城的阿拉伯數學家阿爾・可瓦里茲米(Abu Jafar Muhammad ibn Musa al-Khwarizmi)名字的最後一部份，此數學家將印度所發明的十進位數字記號傳入阿拉伯地區(稍後傳入歐洲並成為現今我們使用的數字記號)，並且著有一本名為"ilm al jabr w'al-muqabala"的書籍，此書有系統地討論一元二次方程的解法，啟發了代數學的發展。此書在 12 世紀被翻譯為拉丁文，名為 Algebra et Almucabala， (Algebra 源自阿拉伯書名中之 al jabr)，而成為代數學(Algebra)一字的由來。「演算法」原為"algorism"，意思是阿拉伯數字的運演算法則，在 18 世紀演變為"algorithm"。歐幾裡得演算法被人們認為是史上第一個演算法。第一次編寫程序是 Ada Byron 於 1842 年為巴貝奇分析機編寫求解解伯努利方程的程序，因此 Ada Byron 被大多數人認為是世界上第一位程式設計師。因為查爾斯·巴貝奇(Charles Babbage)未能完成他的巴貝奇分析機，這個演算法未能在巴貝奇分析機上執行。

周髀算經

圖 1、阿爾・可瓦里茲米

演算法是指完成一個任務所需要的具體步驟和方法。也就是說給定初始狀態或輸入數據，經過電腦程序的有限次運算，能夠得出所要求或期望的終止狀態或輸出數據。

演算法常常含有重複的步驟和一些比較或邏輯判斷。如果一個演算法有缺陷，或不適合於某個問題，執行這個演算法將不會解決這個問題。不同的演算法可能用不同的時間、空間或效率來完成同樣的任務。一個演算法的優劣可以用空間複雜度與時間複雜度來衡量。

1. 輸入：一個演算法必須有零個或多個輸入量。
2. 輸出：一個演算法應有一個或多個輸出量，輸出量是演算法計算的結果。
3. 確定性：演算法的描述必須無歧義，以保證演算法的執行結果是確定的。
4. 有限性：演算法必須在有限步驟內實現。注：此處「有限」不同於數學概念的「有限」，天文數字般的有限對於實際問題並無意義。
5. 有效性：又稱可行性。能夠實現，演算法中描述的操作都是可以通過已經實現的基本運算執行有限次來實現。

設計一個演算法，首先要考慮的就是正確性(correctness)，也就是演算法是否能產生正確的結果，在達到正確性的要求之後，我們就要特別考慮演算法的效能(efficiency)了，也就是要考慮演算法是否能夠有效率的的執行。一般而言，若是數個演算法均能夠產生相同且正確的結果，則我們大都以有效性來衡量其優劣。

計算複雜性理論（Computational complexity theory）是計算理論的一部分，研究計算問題時所需的資源。最常見的資源是時間（要通過多少步才能解決問題）和空間（在解決問題時需要多少記憶體）。

時間複雜度是指在電腦科學與工程領域完成一個演算法所需要的時間，是衡量一個演算法優劣的重要參數。時間複雜度越小，說明該演算法效率越高，則該演算法越有價值。

空間複雜度是指電腦科學領域完成一個演算法所需要占用的存儲空間，一般是輸入參數的函數。它是演算法優劣的重要度量指標，一般來說，空間複雜度越小，演算法越好。我們假設有一個圖靈機來解決某一類語言的某一問題，設有 X 個字（word）屬於這個問題，把 X 放入這個圖靈機的輸入端，這個圖靈機為解決此問題所需要的工作帶格子數總和稱為**空間**。

一般我們使用自然語言(中文或英文等語言)、流程圖(flow chart)、虛擬碼(pseudo code)或高階程式語言(high level programming language)來表示演算法。虛擬碼以一種混雜著自然語言與高階程式語言結構的方式來描述演算法，試圖達到簡潔易讀、容易分析，而且也容易轉換為高階程式語言的目的。

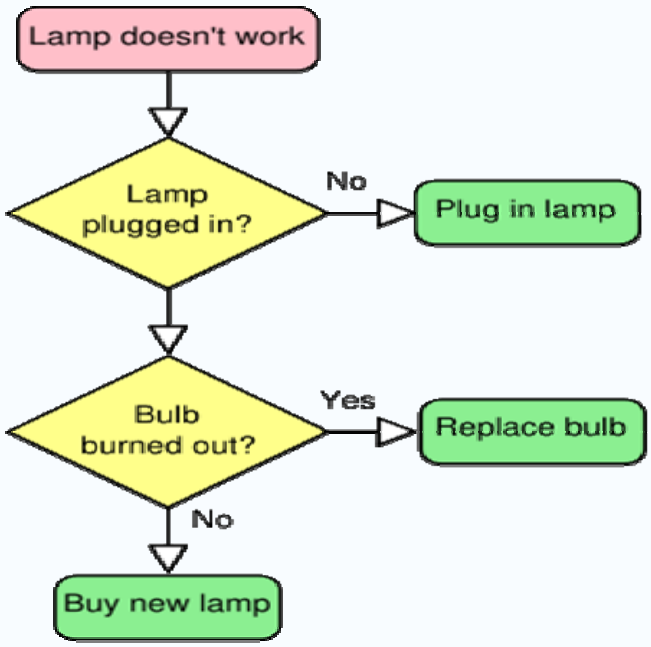

圖 2、流程圖

二、歐幾里德演算法

問題：此演算法假設輸入為兩正整數，並求出其最大公因數。

程序：

步驟 1. 將 M 設為較大數，N 設為較小數。

步驟 2. M 除以 N，餘數為 R。

步驟 3. 如 R 不為 0，則將 M 設為 N，設 N 為 R 並回到步驟 2；否則，最大公因數即是現在指派給 N 的值。

$$GCD(n,m)=\begin{cases} GCD(m.n) & if\ n<m \\ m \quad if\ n\geq m\ and\ n \bmod m=0 \\ GCD(m,n \bmod m) & otherwise \end{cases}$$

```
int gcd(unsigned m, unsigned n)
   { return(0== n? m : gcd(n, m%n)); }
```

```
int gcd2(unsigned m, unsigned n) // Iterative procedure
{
    unsigned temp;
    while (0 != n)
      {    temp = n;      n = m % n;      m = temp;      }
    return m;
}
```

gcd(314159, 271828)

gcd(271828, 42331)

gcd(42331, 17842)

gcd(17842, 6647)

gcd(6647, 4458)

gcd(4458, 2099)

gcd(2099, 350)

gcd(350, 349)

gcd(349, 1)

gcd(1, 0)

三、資料的搜尋

從資料檔案中，尋找符合某特定條件的記錄。」這個動作就叫搜尋。而用來搜尋的條件就稱為「鍵值」，例如我們在電話簿中找某人的電話，那麼這個人的姓名就成為在電話簿中搜尋電話資料的鍵值。 若依資料量大小來區分，搜尋可分為：

1. 內部搜尋：資料量較小的檔案，可以直接全部載入記憶體中進行搜尋。
2. 外部搜尋：資料量大的檔案，無法一次載入記憶體處理，而需使用到輔助記憶體來分次處理。

除了上述的分類方式外，我們還能以搜尋過程中被搜尋的表格或資料是否異動，區分為靜態搜尋(Static Search)及動態搜尋(Dynamic Search)。

1. 靜態搜尋是指資料在搜尋過程中，該搜尋資料不會有增加、刪除、或更新等行為，例如符號表搜尋就屬於一種靜態搜尋。
2. 動態搜尋則是指所搜尋的資料，在搜尋過程中會經常性地增加、刪除、或更新。

影響搜尋時間長短的主要因素

1. 演算法
2. 資料儲存的方式
3. 資料儲存的結構

3.1 循序搜尋法(Sequential Search)

循序搜尋又稱線性搜尋，是一種最簡單的搜尋法。它的方法是將資料一筆一筆的循序搜尋，這很像在走訪陣列一般的從頭找到尾，所以不管資料是否經過排序，都是得從頭到尾走訪過一次。此法的優點是檔案在搜尋前不需作任何的處理與排序，缺點為搜尋速度較慢。

Problem: $x \in S[n]$?
Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*, and a key *x*.
Outputs: location, the location of *x* in *S* (0 if $x \notin S$).
Strategy: The method scans the elements, one after another, until succeeds or it exhausts the range:
Algorithm:

```
void seqsearch1 (int n, Keytype S[ ], Keytype x, Index& location)
{              // index& denote location is an output parameter.
    for (location = 1; location <= n && S[location] != x; location ++) ;
    if (location > n)    location = 0;
}

// Another one
void seqsearch2 (int n, Keytype S[ ], Keytype x, Index& location)
```

```
{               // index& denote location is an output parameter.
    location = 1;
    while (location <= n && S[location] != x)    location++;
    if (location > n)    location = 0;
}

// Another one
void seqsearch3(int n, Keytype S[ ], Keytype x, Index& location)
{
    location = 0;
    while (++location <= n && S[location] != x) ;
    if (location > n)    location = 0;
}

// Error version
void seqsearch4 (int n, Keytype S[ ], Keytype x, Index& location)
{
    location = 1;
    while (location++ <= n && S[location] != x) ;
    if (location > n)    location = 0;
}

// Exact version
void seqsearch5(int n, Keytype S[ ], Keytype x, Index& location)
{
    location = 1;
    while (location <= n && S[location++] != x) ;
    if (location > n)    location = 0;
}
```

這個方法基本上沒有錯，但是可以加以改善，可以利用設定衛兵的方式，省去 if 判斷式，衛兵通常設定在數列最後或是最前方，假設設定在列前方好了（索引 0 的位置），我們從數列後方向前找，如果找到指定的資料時，其索引值不是 0，表示在數列走訪完之前就找到了，在程式的撰寫上，只要使用一個 while 迴圈就可以了。

```
// A better one.
void seqsearch6(const unsigned n, Keytype S[ ], Keytype x, Index& location)
{
    S[n+1] = x;      // Set a sentinel at the end of S.
    location = 0;
    while (S[++location] != x);      // Neglect the checking of list end
    if (location > n)    location = 0;
```

```
}

// A error one.
void seqsearch6(const unsigned n, const Keytype S[ ], Keytype x, Index& location)
{
    S[n+1] = x;     // Set a sentinel at the end of S.
    location = 0;
    while (S[++location] != x);     // Neglect the checking of list end
    if (location > n)    location = 0;
}

// For-loop version.
void seqsearch7(const unsigned n, Keytype S[ ], Keytype x, Index& location)
{
    S[n+1] = x;     // Set a sentinel at the end of S.
    for (location = 1; S[location] != x; ++ location)    ;
    if (location > n)    location = 0;
}

// Function-type version
Index seqsearch8 (const unsigned n, Keytype S[ ], Keytype x)
{
    S[n+1] = x;     // Set a sentinel at the end of S.
    Index location = 0;
    while (S[++location] != x);     // Neglect the checking of list end
    return (location > n)? 0 : location;
}

// A faster one.
Index seqsearch9 (const unsigned n, Keytype S[ ], Keytype x)
{
    S[n+1] = x;             // Set a sentinel at the end of S.
    Index location = 1;
    while (1)
    {
        if (S[location] == x)    break;
        if (S[location+1] == x)    {    location += 1;    break;    }
        if (S[location+2] == x)    {    location += 2;    break;    }
        if (S[location+3] == x)    {    location += 3;    break;    }
        if (S[location+4] == x)    {    location += 4;    break;    }
```

```
            if (S[location+5] == x)  {  location += 5;  break;  }
            if (S[location+6] == x)  {  location += 6;  break;  }
            if (S[location+7] == x)  {  location += 7;  break;  }
            location += 8;
            }
        return (location > n)? 0 : location;
    }
```

Basic operation: the comparison of an item in the array with x.

Worst-case time complexity: x is the last item in the array or x is not in the array.

$$W(n) = n.$$

Best-case time complexity: x is in the first item in the array.

$$B(n) = 1.$$

Average-case time complexity:

(1) $x \in S$

$$A(n) = \sum_{k=1}^{n}\left(k \times \frac{1}{n}\right) = \frac{1}{n}\sum_{k=1}^{n} k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

(2) x may not be in S: Let p to be the probability that x is in the array.

$$A(n) = \sum_{k=1}^{n}\left(k \times \frac{p}{n}\right) + n(1 - p) = \frac{p}{n}\sum_{k=1}^{n} k + n(1 - p) = n\left(1 - \frac{p}{2}\right) + \frac{p}{2}$$

```
// SeqSearch.cpp
#include <stdio.h>
#include <conio.h>
#include <time.h>

typedef int KeyType;
typedef unsigned Index;
#define START 0
#define END -1

void EnterTarget(KeyType*);
void Error(char*);
Index SeqSearch(KeyType List[], Index, KeyType, unsigned*);
void BuildList(KeyType List[], Index*);
float Time(int);
int UserSayYes(void);

void main()
{
    const unsigned MAXLIST = 100;
```

```
    KeyType List[MAXLIST];      KeyType  target;      unsigned  compare;
    Index  top, targetlocation;      float runtime;
    BuildList(List, &top);
    if (-1 == top) return;
    do {
        EnterTarget(&target);
        Time(START);
        compare = 0;
        targetlocation = SeqSearch(List, top, target, &compare);
        runtime = Time(END);
        printf("Successful search after %d comparisons and running time is %f ms\n",
                        compare, runtime);
        if (-1 == targetlocation) printf("\nKey %d not in the list.", target);
        else   printf("\n Target locates at %d-th element.", targetlocation+1);
        printf("\n\nDo you want to execute this program again?");
     } while (UserSayYes()) ;
}

void EnterTarget(KeyType *target)
{
  do {
       printf("\nEnter the number you want to find\n");
       scanf("%d", target);
       if (*target > 0) break;
       Error("The key in number must > 0. Enter target number again");
     } while(1);
}

void Error(char *message)                              /* Display an error message */
{   printf("Error: %s\n", message);   }

void BuildList(KeyType List[], Index *top)
{
    int i =0, c;
    printf("Key in a series of positive integer for database. Terminate with 0.\n");
    scanf("%d", List);
    do {
        scanf("%d", &c);
        if (0 == c)   break;
        else if (c < 0)   { Error("Input data must > 0\n"); *top = -1; break; }
```

```
            else    List[++i] = c;
        } while(1);
        *top = i;
}

Index SeqSearch(KeyType List[], Index top, KeyType target, unsigned *cmp)
{
    Index location = -1;
    *cmp = 1;
    List[top+1] = target;                                   /* Set sentinal */
    while (List[++location] != target)      (*cmp)++;
    return (location > top)? -1 : location;
}

float Time(int flag)      // return runtime (ms)
{
        static float start;    float end;
        if (flag == START) { start = clock();    return 0.0; }
        else { end = clock();    return(end-start)*1000/CLK_TCK; }    // CLK_TCK = 1000
}

int UserSayYes(void)
{
        int c;      // Don't use 'char c'
        printf("(y, n)?");
        do {
                while ((c = getchar()) == '\n') ;           /*Ignore new line character*/
                if (c == 'y' || c == 'Y'|| c == 'n' || c == 'N')
                        return    (c == 'y' || c == 'Y');
                printf("Please respond by typing one of the letters y or n\n");
        } while(1);
}
```

3.2 二元搜尋法(Binary Search)

如果要搜尋的資料已經排序好，則可使用二分法來進行搜尋。二分法是將資料分割成兩等份，再比較鍵值與中間值的大小，如果鍵值小於中間值，可確定要找的資料在前半段的元素，如此分割數次直到找到為止。

Alg. 1.5　Binary search (二元搜尋)

Problem: Is the key *x* in the sorted array *S* of *n* keys?

Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, and a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

Procedure:

```
void binsearch1 (unsigned n, const Keytype S[ ], Keytype x, Index& location)
{
    Index low = 1, high = n, mid;      location = 0;
    while (low <= high && 0 == location)
    {
        mid = (unsigned)(low + high)/2;
        if (x < S[mid])    high = mid-1;
        else if (x > S[mid]) low = mid + 1;
        else       location = mid;
    }
}

// A better one
Index binsearch2 (unsigned n, const Keytype S[ ], Keytype x)
{
    Index low = 0, high = n+1, mid;    // Add sentinels
    while (low+1 != high)
    {
        mid = (unsigned)(low + high)/2;
        if (x <= S[mid])    high = mid;          else    low = mid;
    }
    Index location = high;
    return (location >= n+1 || S[location] != x)? 0 : high;
}
```

Basic operation: the comparison of *x* with *S*[*mid*].

Worst-case time complexity:

$W(n) = W(n/2) + 1;\ W(1) = 1. \quad \Rightarrow \quad W(n) = \lfloor \lg n \rfloor + 1$

分析：

1. 二分法必須事先經過排序，且資料量必須能直接在記憶體中執行。
2. 此法適合用於不需增刪的靜態資料。
3. 循序搜尋法與二元搜尋法的優缺點

搜尋法	優點	缺點
循序搜尋法	資料無需事先排序	搜尋速度慢，沒有效率
二元搜尋法	和循序搜尋法相較起來，其搜尋速度較快	必須事先將資料排序好，且儲存裝置必須能夠直接存取，例如磁帶就不適合二元搜尋法。

3.3 內插搜尋法(Interpolation Search)

內插搜尋法又叫做插補搜尋法，是二元搜尋法的改良版。它是依照資料位置的分佈，利用公式預測資料的所在位置，再以二分法的方式漸漸逼近。使用內插法是假設資料平均分佈在陣列中，而每一筆資料的差距是相當接近或有一定的距離比例。其內插法的公式為：

$$mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$$

插補搜尋法的步驟

1. key 是要尋找的鍵，data[high]、data[low]是剩餘待尋找記錄中的最大值及最小值，對資料筆數為 n，插補搜尋法的步驟如下：
2. 將記錄由小到大的順序給予 1,2,3...n 的編號
3. 令 low=1，high=n
4. 當 low<high 時，重複執行步驟 4 及步驟 5
5. 令 Mid=low + ((key - data[low]) / (data[high] -data[low]))* (high - low)
6.
   (1) 若 key<key[Mid]且 high≠Mid-1 則令 high=Mid-1
   (2) 若 key＝key[Mid]表示成功搜尋到鍵值的位置
   (3) 若 key>key[Mid]且 low≠Mid+1 則令 low=Mid+1

Problem: Determine whether x is in the sorted array of size n.

Inputs: positive integer n, and sorted (nondecreasing order) array of numbers S indexed from 1 to n.

Outputs: the location i of x in S; 0 if x $\notin$ S.

Formula: $mid = low + \left\lfloor \frac{x - S[low]}{S[high] - S[low]} \times (high - low) \right\rfloor$

Procedure:

```
void interpsrch(int n, const number S[], number x, index& i)
{
    index low = 1, high = n, mid;    number denominator;    i = 0;
    if (S[low] ≤ x ≤ S[high])
        while (low <= high && 0 == i)
        {
            denominator = S[high] – S[low];
            if (0 == denominator)    mid = low;
            else    mid = low +⌊((x-S[low])*(high-low))/denominator⌋;
```

```
            if (x < S[mid])      high = mid – 1;
            else if (x == S[mid])    i = mid;
            else low = mid +1;
        }
}
```

分析:

1. 資料需先經過排序。
2. 一般而言，內插搜尋法優於循序搜尋法，而如果資料的分佈愈平均，則搜尋速度愈快，甚至可能第一次就找到資料。此法的時間複雜度取決於資料分佈的情況而定，平均而言優於 O(log n)。