5-2演算法的複雜度

1

演算法

離散數學中有許多一般性的問題：給定一序列的整數，找出之間最大者；給定一個集合，表列出所有的子集合；給定一個整數集合，將其元素依遞增方式排列；給定一個網路，找出兩頂點間的最短路徑等等。在面對這些問題時，第一件事就是將問題轉化成數學模式。

找出適切的數學模型只是求解的一部分。為求出完整的解答，還必須找出求解這類模型之一般化問題的方法。最理想的方式，就是找出一連串的步驟，能趨向所要的解。這種一序列之步驟稱為**演算法**（**algorithm**）。

2

找出一個有限序列中的最大元素

描述找出一個有限整數列中的最大值的演算法。

演算法　找出一個有限序列中的最大元素

procedure $max(a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ **to** n
　　if $max < a_i$ **then** $max := a_i$
{max is the largest element}

a_1	a_2	a_3	a_4	a_5
3	1	6	8	5

i =	2	3	4	5
max=	3	6	8	8

演算法首先將序列中第一項 a_1 指定給變數 max。"for" 迴圈用於逐一檢查序列中的每一項。若檢查之項大於現存於max中的值時，將新的值指定給max。

3

線性搜尋

一般化的搜索問題可描述如下：在不同元素 $a_1, a_2, \ldots, a_n$ 中，找出元素 x，或是判斷其並不在列表之中。問題的解為找出位置 i，使得 $a_i = x$；若 x 不在集合之間，回傳 0。

線性搜索　首先介紹的演算法稱為**線性搜索**（**linear search**）或是**序列搜索**（**sequential search**）。線性搜索由比較 x 與第一項 a_1 開始，若 $x = a_1$ 則回傳 1，表示 x 在第一個位置上。若是 $x \neq a_1$，則與下一項 a_2 比較。若 $x = a_2$ 則回傳 2，表示 x 在第二個位置上。若是 $x \neq a_2$，則繼續向下比較，直到找出 x 所在的位置，或是與表列中所有的元素都不相同為止。若在表列中找不到相同的元素，則令解答為 0。

4

線性搜尋－虛擬碼

演算法　線性搜索演算法

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
　　$i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
{*location* is the subscript of the term that equals x, or is 0 if x is not found}

x	a_1	a_2	a_3	a_4	a_5
8	3	1	6	8	5

i =	1	2	3	4
***location*=**				**4**

5

演算法的複雜度

如何分析演算法的效率？有一種度量效率的方法是在某特定輸入規模時，測量電腦使用該演算法解題所花費的時間。第二種度量方法則是在特定輸入的規模時，觀察電腦使用該演算法解題所需的記憶體。

上述問題皆與演算法的**計算複雜度**（**computational complexity**）有關。分析求解特定規模問題所需要的時間牽涉到演算法的**時間複雜度**（**time complexity**）。分析需要多少電腦記憶體則與演算法的**空間複雜度**（**space complexity**）有關。在實際操作演算法時，考慮演算法時間複雜度與空間複雜度是不容忽視的。很明顯地，了解演算法是否能在一微秒、一分鐘或是幾十億年後才能產生正確解非常重要。同樣地，求解問題所需的記憶體也應該是必須知道的，所以空間複雜度也應納入考量。

空間複雜度與執行演算法的資料結構密切相關。所以暫不考慮空間複雜度，而是將注意力集中在時間複雜度上。

6

時間複雜度

演算法的時間複雜度可以用該演算法在特定輸入規模的情況下，電腦執行的演算次數表示。用以測量時間複雜度所需的運算可能是比較整數大小、整數加法、整數除法或者其他的基本運算。

時間複雜度是以所需的運算次數加以表示，而不是以電腦執行的實際時間來表示。因為不同的電腦執行基本運算時所需的時間並不相同，而且要將所有運算分解為電腦所使用的基本的位元操作是非常複雜的過程。除此之外，目前已知最快速的電腦可以在 10^{-9} 秒內（1 奈秒）執行基本位元運算，但個人電腦可能需要 10^{-6} 秒鐘（1 微秒），兩者執行相同運算的時間相差 1000 倍。

7

找出一個有限序列中的最大元素 - 時間複雜度分析

演算法　找出一個有限序列中的最大元素

```
procedure max(a_1, a_2, ..., a_n: integers)
max := a_1
for i := 2 to n
      if max < a_i then max := a_i
{max is the largest element}
```

既然數值比較為該演算法所使用的基本運算，我們便以比較的次數作為時間複雜度的測度值。

針對表列中的每一項，不斷以下列的兩次比較重複前述程序；一個比較是判斷是否到達表列末端，另一個比較則是判斷要將暫時最大值更新。既然從第二項到第 n 項都要執行兩次比較，而且當 $i = n + 1$ 時還需要一次離開迴圈所需的比較，所以使用這個演算時所執行的比較次數恰為 $2(n - 1) + 1 = 2n + 1$ 次。因此，找出 n 個元素中最大元素的演算法其時間複雜度為 $\Theta(n)$，這是以演算法所執行次數加以測量的結果。

8

線性搜尋演算法 - 時間複雜度分析

演算法 線性搜索演算法

```
procedure linear search(x: integer, a1, a2, ..., an: distinct integers)
i := 1
while (i ≤ n and x ≠ ai)
      i := i + 1
if i ≤ n then location := i
else location := 0
{location is the subscript of the term that equals x, or is 0 if x is not found}
```

演算法所執行的比較次數將會作為時間複雜度的測度值。在演算法迴圈的每一個步驟裡，電腦執行兩次比較：其一是檢查是否到達列表末端，其二是比較 x 與列表的項。最後，在迴圈外還有一次比較。因此，如果 $x = a_i$，則使用了 $2i + 1$ 次比較。當元素不在表列中時，最多需要比較 $2n + 2$ 次。因此，線性搜索至多執行 $O(n)$ 次的比較。

9

最差狀況與平均狀況的複雜度

最差狀況的複雜度 上例的複雜度分析是**最差狀況（worst-case）**的分析。所謂演算法的最差狀況，是指為了使用特定演算法求解已知輸入規模的問題時所需要的最多運算次數。最差狀況分析可以告訴我們需要多少次的運算才能保證演算法可以產生解。

平均狀況的複雜度 除了最差狀況的分析之外，還有一種重要的複雜度分析，稱為**平均狀況（average-case）**分析。這種分析方式可以求出，針對輸入規模已知的所有問題而言，運算的平均次數是多少。平均狀況時間複雜度分析通常比最差狀況還要複雜得多。然而，對於線性搜索演算法而言，平均狀況分析卻不難完成。

10

線性搜尋演算法 - 平均狀況複雜度

演算法　線性搜索演算法

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
　　$i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$

描述線性搜索演算法的平均狀況，假設表列中含有元素 x。

已知元素 x 確實屬於表列時，有 n 種可能的輸入。如果 x 是表列的第一項，必須執行三次比較。如果x是表列的第二項，必須執行五次比較。一般而言，如果 x 是表列中的第 i 項，需要的比較總數是 $2i + 1$。所以比較次數的平均值是 $\frac{3+5+7+\cdots+(2n+1)}{n} = \frac{2(1+2+3+\cdots+n)+n}{n} = \frac{2[n(n+1)/2]}{n} + 1$ $= n + 2$，也就是$\Theta(n)$。