程式設計入門

官大智 教授　中山大學 資訊工程學系

September 11, 2005

1　簡介

由於技術的進步, 計算機的功能大爲提高, 現今的計算機每秒鐘可以處理數千萬甚至上億個指令已不算稀奇. 但是不管計算機的計算速度有多快, 它只能依照事先安排好的指令去執行, 無法自行設計解決問題的各種方法. 因此想要利用計算機去解決問題, 必須設計解決此問題的程式.

設計程式之前, 先要有解決問題的方法, 或叫做演算法. 除了演算法之外, 資料的表示和儲存的方法也是很重要的. 這門學問叫做資料結構. 好的資料結構可以方便各種操作, 使演算法更簡單, 更快速. 資料結構和演算法可以說是程式設計的理論基礎.

我們將用實例來說明資料結構和演算法的設計, 這些例子包括: 輾轉相除法, 由日期計算星期幾, 搜尋, 排序, 解非線性方程式的根, 解線性方程式組, 等.

程式設計好之後, 還要有編譯器將這些程式翻譯成計算機可以執行的機器語言. 我們將用類似 PASCAL 的語言來描述演算法. 若使用 PASCAL 編譯器, 則很容易將這些例子寫成 PASCAL 程式. 若選用別的編譯器, 也很容易將這些演算法轉換成該語言的程式.

2　演算法簡介

演算法英文叫做 algorithm, 依照 Webster 字典的定義, 演算法就是:

> any special method of sloving a certain kind of problem.

簡單地說, 演算法就是由輸入資料求得輸出資料的方法. 例如求兩個正整數 a 和 b 的最大公約數可以用輾轉相除法. 因此, 我們稱輾轉相除法爲求兩個正整數 a 和 b 的最大公約數的演算法. 我們可以將演算法視爲一組計算步驟, 每個步驟所做的事必需非常明確. 同時對任一組輸入資料, 這個演算法必須在執行有限個步驟之後得到答案. 我們先用輾轉相除法爲例, 介紹演算法的設計和分析.

輾轉相除法 (Euclid Method) 是一個求得兩正整數的最大公約數的演算法. 其方法爲: 任意給定兩正整數 a 和 b, 若 a 能被 b 整除, 則 a, b 兩數的最大公約數就是 b. 若不能被 b 整除, 則先求出 a 除以 b 的餘數 r. a 和 b 的最大公約數就等於 b 和 r 的最大公約數, 亦即以 b 取代 a, 以 r 取代 b 反覆執行以上的計算, 直到餘數爲 0 時停止.

以上的文字說明, 不僅繁複, 而且會因爲自然語言的不確定性, 可能會產生不同的解釋的情況. 因此演算法通常不用一般的文字來說明. 以前較常用來表示演算法的方法是流程圖, 以輾轉相除法爲例, 其流程圖可以用圖 1 來表示.

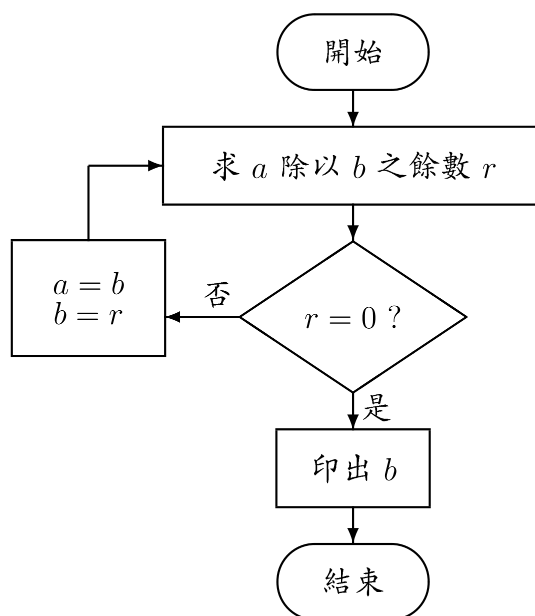

Figure 1: 輾轉相除法流程圖

由圖 1 的流程圖可以看出, 每個演算法都有一個固定開始的地方, 而且每一個步驟所做的事情必須要很明確.

除此之外, 對任一個合法的輸入資料, 此演算法必須要在有限的計算步驟結束. 以輾轉相除法爲例, r 是 a 除以 b 的餘數, 因此其值必爲正數而且小於 b. 也就是說每做一次計算 a 除以 b 的餘數 r 的時候, r 的值會越來越小, 但一定會大於或等於 0. 由此可知, 經過有限次數的計算之後 r 的值必會等於 0. 當然需要計算多少次是和 a 與 b 的值有關的, 但是不管 a 和 b 之值爲何, 經過反覆運算之後此演算法必定會印出最大公約數, 而結束運算.

總之, 一個正確的演算法必需要有下列特性:

1. 每個計算步驟必需要明確.

2. 對任一合法的輸入資料, 必需要在有限的計算步驟求出答案.

演算法除了必需符合上述的兩個條件之外, 對於演算法本身的描述也應該是有限的. 也就是說, 只能用有限個文字或符號來描述演算法.

由於演算法每個步驟的確定性, 對於任一個合法的輸入資料, 其計算出來的結果一定是唯一的. 因此可以將演算法看成爲一個函數. 此函數的定義域就是所有合法輸入資料所成的集合, 對任一合法的輸入資料, 其所對應的函數值就是此演算法以此輸入資料所計算得到的結果.

要注意, 能夠被計算的函數, 其定義域與值域的元素都必需能夠用有限個符號來表示. 例如, 若用十進位小數來表示 1/3 則需要有無窮多個位數. 但是 1/3 可以用兩個數 $p = 1$ 和 $q = 3$ 來表示. 但對任一實數而言此點是做不到的, 因此我們限制在討論整數的函數. 事實上, 所有的實數計算在計算機中都無法執行, 只能取其近似值而已!

通常演算法是設計來解決“同一類型”的問題, 因此合法的輸入資料有無限多組. 演算法不宜只針對某些特例來設計. 例如解一元二次方程式 $ax^2 + bx + c = 0$ 的根的演算法必需有能力處理任何 a, b, c 之值, 而不宜只針對 $2x^2 + 5y + 3 = 0$ 去設計演算法.

不過, 爲了簡單起見, 有時候也會對輸入的資料做一些限制. 例如, 可以限制這個程式只解 $ax^2 + bx + c = 0$ 中 $b^2 - 4ac \geq 0$ 的情形, 而不考慮有虛

數根的情況. 總之, 在設計演算法之前, 必需要把問題定義清楚. 然後再根據問題來設計演算法.

演算法的每個步驟必須很明確. 在理論上, 我們先要用數學的方法定義計算機. 然後, 我們就可以說, 一個明確的步驟就是指計算機能夠執行的命令. 我們簡化了這個步驟, 就以現有的計算機, 例如: 個人電腦或超級電腦都可以, 做爲我們的計算機數學模型. 不過, 通常我們不是用計算機的機器語言來撰寫程式, 而且使用組合語言的機會也越來越少了. 我們也可以選用某一種高階語言, 例如 PASCAL, C, BASIC, FORTRAN, 等, 做爲我們的計算機數學模型. 一個明確的步驟就是這個語言所能夠表達的運算. 但是這些語言的語法在用來描述演算法的時後, 也許會覺得太複雜了, 在不會產生混淆的情況下, 用自然語言來描述演算法, 也是可以被接受的. 以輾轉相除法求兩個證整數 a 和 b 的最大公因數爲例, 說明如圖 2.

輸入: 正整數 a 和 b.

輸出: a 和 b 的最大公因數, c.

方法:

1. 讀入 a 和 b
2. 計算 a 除以 b 的餘數 r
3. 若 $r = 0$ 則 $c = b$; 結束
4. $a = b$
5. $b = r$
6. 回到第 2 步

Figure 2: 輾轉相除法求兩個正整數 a 和 b 的最大公因數 c

以前常用流程圖來表示演算法. 用流程圖來表示演算法可以清楚的看出整個演算法的計算程序. 一個複雜的運算程序可以用流程圖來表示. 但是, 設計一個演算法的時候, 希望計算程序結構化. 計算程序複雜的演算法較不容易讀懂, 要修改也可能牽一髮動全身, 不容易下手. 因此, 演算法的控制結

構應該簡化. 基本上, 下列的控制結構是必需的:

1. 順序運算
   即第一個步驟做完之後做第二個步驟, 第二個步驟做完後做第三個步驟, 等等.

2. **if** C_1 **then** S_1 **else** S_2 ;
   先判斷 C_1 是否正確, 若是正確則執行 S_1, 否則執行 S_2 .

3. **while** C_1 **do** S_1 **end**
   判斷 C_1 是否爲正確, 若是正確即執行 S_1, 而且做完之後再回頭判斷 C_1 是否正確, 若是正確則再執行 S_1. 如此反覆執行, 直到 C_1 爲不正確時即停止. 在 S_1 中應該會有適當的計算使得 C_1 會因爲執行 S_1 而改變其正確與否, 否則就會造成無窮的迴圈.

用以上三種控制結構, 來描述輾轉相除法, 即得到如圖 3 之演算法:

輸入: 正整數 a 和 b.

輸出: a 和 b 的最大公因數, c.

方法:

讀入 a 和 b;

while $b > 0$ **do begin**

　　計算 a 除以 b 的餘數 r;

　　$a = b$;

　　$b = r$;

end;

$c = a$;

Figure 3: 輾轉相除法

雖然只有三種不同的控制結構, 但是經過反覆的使用, 可以設計出各種

複雜的演算方法. 而且也可以証明所有計算機能執行的程式都可以利用上面三種控制結構來描述.

以上三種控制結構爲最基本的控制結構, 有些程式語言, 例如 C 及 PASCAL 提供類似上述三種控制結構. 如果演算法是由上述三種控制結構來描述, 而且選用 C 或是 PASCAL 語言, 則撰寫程式時只要再加上資料結構的描述便可以將演算法轉換成程式. 若是選用別的語言, 例如 1977 年以前的 FORTRAN, 則因爲舊版的 FORTRAN 並沒有直接提供這些控制結構, 因此撰寫程式時還需要一些轉換的工作.

爲了方便起見, 有時候我們也會增加一些其他的控制結構來描述演算法, 這些新增加的控制結構都可以以上列三種基本控制結構去改寫.

3 資料的表示方法

所有的要處理的資料都要以某種形式儲存在計算機的記憶體中. 記憶體的最小單元是 bit, 每一 bit 只能代表 0 或 1, 因此只能記錄兩種不同的資料. 通常將 8 個 bit 集在一起, 成爲 byte. 每一個 byte 因爲有 8 個 bit, 可以紀錄 256 種不同的資料. 英文大小寫字母, 加上標點符號, 運算符號, 等等不會超過 256 個, 因此常用一個 byte 來表示. 但是中文字有好幾萬個, 就要用二個 byte, 或者更多個 byte 來表示.

以上說的是文字的表示方法. 數字在計算機中通常用 4 個 byte, 8 個 byte, 或者更多個 byte 來表示, 而且整數, 小數各有不同的表示方式, 設計程式的時候必須要注意, 不可將它們混淆.

正整數的表示方法比較簡單. 由於每一 bit 能夠代表 0 或 1, 將整數化成二進位數就可以了. 因此, 4 個 byte 的正整數最小是 0, 最大是 $2^{32} - 1 = 4294967295$. 若此數不夠大, 有些時候可以使用 8 個 byte, 最大的值就可以到 $2^{64} - 1 = 18446744073709551615$. 如果要處理的數比 18446744073709551615 還大, 就要自行用其他的方法來處理.

整數因爲有正有負, 不能只化成二進位來表示. 它的表示方法有很多種, 比較常用的是補數法. 正數時, 它和正整數的表示法相同, 負數時, 假設此數爲 x, 則用 $2^m + x + 1$ 來表示 x, 其中 m 是一共有幾個 bit 來表示這個數.

注意用此法所表示的數必須介於 -2^{m-1} 和 $2^{m-1}-1$ 之間的整數. 最高 bit 爲 1 的是負數. 負數也可以先看成正數, 再將 0 變 1, 1 變 0, 最後再加 1 來完成.

實數在計算機中是無法表示的, 只能表示有限位數的小數. 選定一個基底 β, 通常 $\beta = 2^k$, 例如 $\beta = 2$ 或 $\beta = 16$. 假設此小數爲 x. 先將 x 表示成 $\pm y \cdot \beta^\alpha$, 其中 y 爲介於 0 與 1 之間的數. 因此, 小數爲 x 被分成三個部分來儲存. 通常先儲存正負號, 再儲存 α, 再儲存 y. 正負號用 0 表示正, 1 表示負, 只要一個 bit. 用多少個 bit 來儲存 α 和 y 在不同的機器上不盡相同, 所以能表示的數值範圍也不一樣.

以上所說的是基本的資料, 大部份現今計算機的硬體都可以提供這些基本資料的儲存和運算. 如果要處理的資料並不是這些基本資料, 就要靠軟體來完成. 例如: 複數可以用兩個實數 (x, y) 來表示, 其中 x 代表實部 y 代表虛部. 因此, 在計算機中可以用兩個小數來表示實數的近似值. 由於複數的儲存和運算並非由計算機的硬體直接提供, 做複數的運算就要靠軟體來完成. 也就是說, 每一個實數的運算都要自己要寫一段程式來做. 這樣不僅比較麻煩, 而且計算的速度也比較慢. 不過, 有些專爲數值計算而設計的編譯器, 如 FORTRAN 可以提供複數的資料型態, 在寫程式的時候可以直接將複數看成是一般的數來處理. 這樣雖然在此寫程式的時候比較方便, 但是它的運算還是要靠軟體來執行, 速度慢的問題無法因此而改善.

現在流行的 object-oriented programming 可以將任何一個要處理的資料, 例如: 日期, 和處理的方法, 例如: 計算此日期是星期幾, 視爲一個 object, 這樣可以更方便撰寫程式. 但是, 這並不表示我們不需要學習資料結構, object-oriented programming 只能提供我們一個更好的工具, 將我們設計好的資料結構和演算方法以更好的方式來處理.

4 求餘數與計算星期幾

給定一個日期, 問你是星期幾, 你會回答嗎? 現在讓我們來學習計算星期幾的方法. 在這之前, 我們先介紹一些符號.

1. $x \equiv y \pmod{n}$ 表示 x 和 y 除以 n 的餘數是相同的. 也可以說, $x - y$

能夠被 n 整除.

2. $x \bmod n = r$ 表示 x 除以 n 的餘數是 r. 也就是說, $x \bmod n$ 是求 x 除以 n 的餘數.

3. $\lfloor x \rfloor$ 代表不大於 x 之最大整數, 例如: $\lfloor 3.2 \rfloor = 3$, $\lfloor 6.9 \rfloor = 6$, $\lfloor 9 \rfloor = 9$. 當 x 的值是負數的時候要注意, $\lfloor -1.25 \rfloor = -2$, 不是 -1.

我們先考慮月份和日期. 若知道每月第一天是星期幾, 就可以推得當月的任何一天是星期幾. 我們先看每月一日是星期幾的變化情形. 一月有 31 天, $31 \bmod 7 = 3$, 我們可以推得二月一日的星期是一月一日的星期加 3. 平年的二月有 28 天, $28 \bmod 7 = 0$, 我們可以推得三月一日的星期和二月一日的星期是一樣的. 三月有 31 天, $31 \bmod 7 = 3$, 我們可以推得四月一日的星期是三月一日的星期加 3. 四月有 30 天, $30 \bmod 7 = 2$, 我們可以推得五月一日的星期是四月一日的星期加 2. 依此類推, 我們可以建立一個表, 來存這些值. 但是, 我們不想這樣做, 我們想找一個函數, 可以由當年的第一天推得當年每個月第一天應該是星期幾.

由於閏年的二月有 29 天, 我們將月份重排, 將二月排到最後面, 以方便計算. 令 $m = 1$ 代表三月, $m = 2$ 代表四月, $m = 3$ 代表五月, $m = 4$ 代表六月, $m = 5$ 代表七月, $m = 6$ 代表八月, $m = 7$ 代表九月, $m = 8$ 代表十月, $m = 9$ 代表十一月, $m = 10$ 代表十二月, $m = 11$ 代表次年一月, $m = 12$ 代表次年二月. 我們發現,

$$f(m) = \lfloor (13m - 1)/5 \rfloor - 2$$

剛好能滿足因考慮月份需加上的天數, 如圖 4 所示.

有了 $f(m) = \lfloor (13m - 1)/5 \rfloor$ 之後, 就很容易計算當年度某日的星期. 例如, 2000 年 3 月 1 日 是星期 3, 因此, 2000 年 $m + 2$ 月 d 日的星期可以用下列公式來計算.

$$\{2 + d + \lfloor (13m - 1)/5 \rfloor - 2\} \bmod 7$$

上式中, $2 + d$ 是代表2000 年 3 月 1 日是星期 3. 舉幾個例子來試試看:

m 的值	月份	天數	$f(m)$	$f(m)-f(m-1)$
$m=1$	三月	31	0	
$m=2$	四月	30	3	3
$m=3$	五月	31	5	2
$m=4$	六月	30	8	3
$m=5$	七月	31	10	2
$m=6$	八月	31	13	3
$m=7$	九月	30	16	3
$m=8$	十月	31	18	2
$m=9$	十一月	30	21	3
$m=10$	十二月	31	23	2
$m=11$	次年一月	31	26	3
$m=12$	次年二月	28/29	29	3

Figure 4: $f(m)=\lfloor(13m-1)/5\rfloor-2$

1. 2000 年 8 月 1 日, $m=6$, $d=1$.

$$\begin{aligned} & \{2+1+\lfloor(13(6)-1)/5\rfloor-2\} \bmod 7 \\ = & \{\lfloor 15.4\rfloor+1\} \bmod 7 \\ = & \{15+1\} \bmod 7 \\ = & 16 \bmod 7 = 2. \end{aligned}$$

因此, 2000 年 8 月 1 日是星期 2.

2. 2000 年 3 月 29 日, $m=1$, $d=29$.

$$\begin{aligned} & \{2+29+\lfloor(13(1)-1)/5\rfloor-2\} \bmod 7 \\ = & \{\lfloor 2.6\rfloor+29\} \bmod 7 \\ = & \{2+29\} \bmod 7 \\ = & 31 \bmod 7 = 3. \end{aligned}$$

因此, 2000 年 3 月 29 日是星期 3.

3. 2000 年 9 月 28 日, $m=7$, $d=28$.

$$\{2+28+\lfloor(13(7)-1)/5\rfloor-2\} \bmod 7$$

$$
\begin{aligned}
&= \{\lfloor 18 \rfloor + 28\} \bmod 7 \\
&= \{18 + 28\} \bmod 7 \\
&= 46 \bmod 7 = 4.
\end{aligned}
$$

因此, 2000 年 9 月 28 日是星期 4.

還有, 2001 年 1 月和 2 月中的日期也可以用上述的公式來計算星期幾.

1. 2001 年 1 月 1 日, $m = 11$, $d = 1$.

$$
\begin{aligned}
& \{2 + 1 + \lfloor (13(11) - 1)/5 \rfloor - 2\} \bmod 7 \\
&= \{\lfloor 28.4 \rfloor + 1\} \bmod 7 \\
&= \{28 + 1\} \bmod 7 \\
&= 29 \bmod 7 = 1.
\end{aligned}
$$

因此, 2001 年 1 月 1 日是星期 1.

2. 2001 年 2 月 18 日, $m = 12$, $d = 18$.

$$
\begin{aligned}
& \{2 + 18 + \lfloor (13(12) - 1)/5 \rfloor - 2\} \bmod 7 \\
&= \{\lfloor 31 \rfloor + 18\} \bmod 7 \\
&= \{31 + 18\} \bmod 7 \\
&= 49 \bmod 7 = 0.
\end{aligned}
$$

因此, 2001 年 2 月 18 日是星期 0, 也就是星期日.

由以上的例子可以得到一個結論, 只要將每年的 3 月 1 日是星期幾製作成一個表, 就可以計算某年某月某日是星期幾. 但是, 假如你的程式要能夠計算很多的年份, 這個表就會很大. 而且, 如何決定某年的 3 月 1 日是星期幾也是問題. 我們必須尋找更好的方法.

平年有 365 天, $365 \equiv 1 \bmod 7$ 因此推得次年同一天的星期會加 1. 閏年有 366 天, $366 \equiv 2 \bmod 7$ 因此推得次年同一天的星期會加 2. 閏年的規則是: 公元年數能夠被 4 整除但是不能被 100 整除的爲閏年, 其餘爲平年, 但

是公元年數能夠被 400 整除的也是閏年. 我們將公元年數分成兩部分, 假設 c 爲世紀數, y 爲公元年數之末二位數. 假設公元 1600 年 3 月 1 日, 即 $c = 16$, $y = 0$, $m = 1$, $d = 1$, 爲星期 t. 令 $g(c, y) =$

$$\{t + (100c + y - 1600) + \lfloor (100c + y - 1600)/4 \rfloor - \lfloor 3(c - 15)/4 \rfloor\} \bmod 7$$

則 $g(c, y)$ 表示公元 1600 年之後, $100c + y$ 年 3 月 1 日的星期. 爲了方便計算起見, 可以將 $g(c, y)$ 化簡爲

$$g(c, y) = \{t - 2c + \lfloor c/4 \rfloor + y + \lfloor y/4 \rfloor\} \bmod 7$$

上式化簡過程中, 用到 $-\lfloor 3(c - 15)/4 \rfloor = -\lfloor c - 11 - c/4 - 1/4 \rfloor = 11 - c - \lfloor -(c + 1)/4 \rfloor = 11 - c + \lfloor c/4 \rfloor + 1$.

綜合年, 月, 日之後, 星期的計算公式爲:

$$\{t - 2c + \lfloor c/4 \rfloor + y + \lfloor y/4 \rfloor + d + \lfloor (13m - 1)/5 \rfloor - 2\} \bmod 7$$

在上式中, t 的值尚未決定. 要翻 1600 年的日曆似乎不可能. 但是, 我們可以用 2000 年 3 月 1 日是星期 3 來推算 t 的值. 將 $c = 20$, $y = 0$, $m = 1$, $d = 1$, 代入,

$$\begin{aligned} & t - 2(20) + \lfloor 20/4 \rfloor + 0 + \lfloor 0/4 \rfloor + \lfloor (13(1) - 1)/5 \rfloor - 2 + 1 \\ = \; & t - 40 + 5 + \lfloor 2.4 \rfloor - 1 \\ = \; & t - 34 \end{aligned}$$

由 $t - 34 \equiv 3 \pmod 7$ 得到 $t \equiv 3 + 34 \pmod 7$, 也就是說, $t = 2$. 最後, 由年, 月, 日, 來計算星期的公式爲:

$$\{d - 2c + \lfloor c/4 \rfloor + y + \lfloor y/4 \rfloor + \lfloor (13m - 1)/5 \rfloor\} \bmod 7$$

舉 2000 年 10 月 10 日爲例, 將 $c = 20$, $y = 0$, $m = 8$, $d = 10$, 帶入得

$$\begin{aligned} & 10 - 2(20) + \lfloor 20/4 \rfloor + 0 + \lfloor 0/4 \rfloor + \lfloor (13(8) - 1)/5 \rfloor \\ = \; & 10 - 40 + 5 + \lfloor 20.6 \rfloor \\ = \; & -5, \end{aligned}$$

$-5 \bmod 7 = 2$, 所以2000 年 10 月 10 日是星期 2. 在舉 2001 年 1 月 1 日爲例, 將 $c = 20$, $y = 0$, $m = 11$, $d = 1$, 帶入得

$$\begin{aligned} & 1 - 2(20) + \lfloor 20/4 \rfloor + 0 + \lfloor 0/4 \rfloor + \lfloor (13(11) - 1)/5 \rfloor \\ = & \ 1 - 40 + 5 + \lfloor 28.4 \rfloor \\ = & \ -6, \end{aligned}$$

$-6 \bmod 7 = 1$, 所以 2001 年 1 月 1 日是星期 1.

根據上面的公式, 由日期計算星期幾的演算法如 5.

輸入: 年 Y 月 M 日 D.

輸出: w, Y 年 M 月 D 日的星期.

方法:

讀入 Y, M, 和 D;

$y = Y$; $m = M - 2$; $d = D$;

if $m < 0$ **then begin**

　　$y = y - 1$; $m = m + 12$;

end;

$c = y/100$; $y = y \bmod 100$;

$w = \{d - 2c + \lfloor c/4 \rfloor + y + \lfloor y/4 \rfloor + \lfloor (13m - 1)/5 \rfloor\} \bmod 7$;

if $w < 0$ **then** $w = w + 7$;

Figure 5: 由日期計算星期幾

注意, 用上述的公式計算星期幾時, 在 20 世紀前後都沒有問題. 但是在計算較爲古老的日期時, 必須考慮到曆法變更的的問題. 例如, 1752 年的 9 月只有 19 天, 其中 9 月 3 日到 9 月 13 日不見了. 地球的自轉每天慢大約千分之一秒, 現在我們使用的方法是增加閏秒來調整. 說不定過了足夠常的時間之後, 需要調整閏年的計算方式, 或使用其他的曆法也說不定. 但是這個公式至少可以使用到幾個世紀之後是沒有問題的.

5 非線性方程式的解法

現在考慮一元 n 次的方程式 $f(x)$, 若 $n=1$, 則 $f(x)=ax+b$, 求 $f(x)=0$ 的根很容易, 即 $x=-\frac{b}{a}$. 當 $n=2$ 的時候, $f(x)=ax^2+bx+c$, 其 $f(x)=0$ 之根爲 $x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$, 也爲大家所熟悉. 但當 n 很大時, 或是 $f(x)$ 不是 x 的多項式時, $f(x)=0$ 的根就變成較難求得. 以下我們將探討如何用各種計算方法, 求得 $f(x)=0$ 的根的近似解.

5.1 Bisection Method (半分法)

半分法的原理是: 如果 $f(x)$ 在 $[a,b]$ 之間連續且 $f(a)\cdot f(b)\le 0$, 則在 $[a,b]$ 之間必有 $f(x)=0$ 的根. 以 $f(x)=x^3-x-1$ 爲例 $f(1)=-1$, $f(2)=5$, 因此必有一根落在 $[1,2]$ 之中. 若取 1 與 2 之中點 1.5, 計算 $f(1.5)=0.875$, 因此我們可以推論必有一根落在 $[1,1.5]$ 之中. 再取 1 與 1.5 之中點 1.25, 計算 $f(1.25)=-0.296$, 因此我們可以推論必有一根落在 $[1.25,1.5]$ 區間之中. 如此反覆計算, 直到 $f(x)$ 之根所在之區間已經小到我們所需要的精確度爲止. 由以上之說明, 可以得到如圖 6 的演算法:

輸入: $[a,b]$ 區間, 在 $[a,b]$ 區間爲連續的函數 $f(x)$ 且 $f(a)\cdot f(b)\le 0$, 以及精確度 $\varepsilon>0$

輸出: $[a,b]$ 之子區間 $[x,y]$ 包含 $f(x)$ 之根, 且 $y-x\le\varepsilon$

方法:

$x=a$; $y=b$;

while $y-x>\varepsilon$ **do begin**

  $m=(y+x)/2$;

  if $f(x)\cdot f(m)\le 0$ **then** $y=m$ **else** $x=m$;

end;

Figure 6: Bisection Method

5.2 Newton's method (牛頓法)

取 $f(x)$ 圖形上的兩點, $(x_{n-2}, f(x_{n-2}))$ 和 $(x_{n-1}, f(x_{n-1}))$. 通過這兩點, 做一條線, 此線稱爲割線. 此割線與 x-軸的交點 x_n 可以由下列公式計算:

$$x_n = \frac{f(x_{n-1})x_{n-2} - f(x_{n-2})x_{n-1}}{f(x_{n-1}) - f(x_{n-2})}$$

若將計算 x_n 的方法改寫一下:

$$\begin{aligned} x_n &= x_{n-1} - f(x_{n-1}) \cdot \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})} \\ &= x_{n-1} - \frac{f(x_{n-1})}{\frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}} \end{aligned}$$

其中 $\frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}$ 是割線的斜率, 也就是通過 $(x_{n-2}, f(x_{n-2}))$ 和 $(x_{n-1}, f(x_{n-1}))$ 兩點的直線的斜率, 而 x_n 即爲此割線與橫軸的交點.

當 x_{n-1} 和 x_{n-2} 很接近, 亦即 $|x_{n-1} - x_{n-2}| = 0$ 時, $\frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}$ 可以用 $f'(x_{n-1})$ 來代替, 因此,

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

此即爲牛頓法, 其演算法如圖 7.

在計算 x_n 的時候, 只需要用到 x_{n-1} 之值, 因此不需要將所有 $x_0, x_1, \cdots$ 的值保留下來. 若是採用 C, FORTRAN, PASCAL, BASIC 等高階語言, 直接寫

$$x = x - \frac{f(x)}{f'(x)}$$

即可將原來的 x 值取代成新的 x 值.

以上所介紹的二種求 $f(x) = 0$ 之根的方法, 以 $f(x) = x - 0.2\sin x - 0.5$ 在 $[0.5, 1.0]$ 的根爲例說明 bisection method 和 Newton's method 之結果列於圖 8.

由上表可以看出 Newton's method 求得根之收斂速度比較快. 但是要注意, 若初值 (即 x_0) 選擇不當時 Newton's method 可能不會收斂. 因此, 解非線性方程式較爲可行的方法是先用 bisection method 或其他計算較穩定的方法先求得根之近似值, 再用 Newton's method 求得更精確的解.

輸入: x_0 連續且在 x_0 可微分之函數 $f(x)$, 以及精確度 $\varepsilon_0 > 0$

輸出: x_n 使得 $|f(x_n)| \leq \varepsilon$

方法:

$n = 0;$

while $|f(x_n)| > \varepsilon$ **do begin**

$$n = n + 1;$$
$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})};$$

end

Figure 7: Newton's method

n	bisection method	Newton's method
0	0.75	0.5
1	0.625	0.61629718
2	0.5625	0.61546820
3	0.59375	0.61546816
4	0.609375	
5	0.6171875	
6	0.61328125	
	⋮	
19	0.61546850	

Figure 8: 求非線性方程式之根的收斂速度

6 線性聯立方程式的解法

一個 n 元的線性聯立方程式可以寫成:

$$\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\
&\vdots \\
a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n
\end{aligned}$$

求得上述聯立方程式的方法之一是用高斯消去法, 亦即將第 (1) 式乘上 $-\frac{a_{i,1}}{a_{1,1}}$, 再與第 i 式相加. 當 $i=2,3,\ldots,n$ 做完之後, 除了第一式之外, 其他各式之第一項即變爲 0,

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a'_{2,2}x_2 + \cdots + a'_{2,n}x_n &= b'_2 \\ a'_{3,2}x_2 + \cdots + a'_{3,n}x_n &= b'_3 \\ \vdots &\quad \vdots \\ a'_{n,2}x_2 + \cdots + a'_{n,n}x_n &= b'_n \end{aligned}$$

在上式的第二式到第 n 式只包含 $n-1$ 個變數 $x_2, x_3, \cdots, x_n$. 然後反覆應用以上所說的方法, 到此 $n-1$ 個線性聯立方程式. 如此將使第三式到第 n 式變成只有 $x_3, x_4, \cdots, x_n, n-2$ 個變數. 此法反覆應用 $n-1$ 次之後, 可以得到下列之方程式:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a'_{2,2}x_2 + \cdots + a'_{2,n}x_n &= b'_2 \\ &\vdots \\ a^{(n-1)}_{n,n}x_n &= b^{(n-1)}_n \end{aligned}$$

以上計算過程中, 並不需要將個個數值一一記錄下來. 爲了方便起見, 我們將用 $a_{i,j}$ 來取代 $a^{(k)}_{i,j}$. 若一組聯立方程式已化簡爲上述之形式則其求解就變成一件很簡單的工作, 首先求 x_n 之值爲 $b_n/a_{n,n}$. 求出 x_n 之後可以得到 x_{n-1} 之值爲 $\frac{b_{n-1} - a_{n-1,n}\cdot x_n}{a_{n-1,n-1}}$. 如此反覆求出 $x_n, x_{n-1}, \cdots, x_{n-k}$ 之後, 可以求得

$$x_{n-k-1} = \frac{b_{n-k-1} - \sum_{j=n-k}^{n} a_{n-k-1,j}\cdot x_j}{a_{n-k-1,n-k-1}}$$

也就是說,

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} a_{i,j}\cdot x_j}{a_{i,i}}, \quad i = n, n-1, \cdots, 1$$

此法稱之爲高斯消去法 (Gauss elimination) 解線性聯立方程式.

爲方便起見, 令矩陣 W 爲一個 $n \times (n+1)$ 之矩陣, 其值爲

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & & \ddots & & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{pmatrix}$$

在做第 i 次高斯消去法的時候, 需要用到 a_{ii} 當除數. 因此 a_{ii} 不能爲 0. 若此時 $a_{ii} = 0$, 則必需找到另外不爲 0 之方程式並與此方程式交換. 爲了方便起見, 用一個 n 維向量 P 來存第 i 個方程式應該放在那裡, 而不直接做交換的動作. 因此所有的 i 均應用 $P(i)$ 來取代, 例如, a_{ij} 變成 $a_{P(i),j}, b_i$ 變成 $b_{P(i)}$. 圖 9 的演算法用文字來描述的部分讀者可自行化爲用 **if-then-else**, **while** 等控制結構.

輸入: $n \times n+1$ 矩陣 $W=[Ab]$.

輸出: n 維向量 x, 使得 $Ax=b$.

方法:

for $i=1,2,\cdots,n$ **do** $P(i)=i$;

for $k=1,2,\cdots,n-1$ **do begin**

find the smallest $j \geq k$ such that $W_{P(j),k} \neq 0$;

if no such j exist, **then stop**; **else** exchange the value of $P(k)$ and $P(j)$;

for $i=k+1,k+2,\cdots,n$ **do begin**

$m=W_{P(i),k}/W_{P(k),k}$;

$W_{P(i),k}=m$;

for $j=k+1,k+2,\cdots,n+1$ **do begin**;

$W_{P(i),j}=W_{P(i),j}-m\cdot W_{P(k),j}$;

end;

end;

end;

if $W_{P(n),n}=0$ **then stop**;

for $i=n,n-1,\cdots,1$ **do begin**

$$x_i=\frac{b_{P(i)}-\sum_{j=i+1}^{n} a_{P(i),j}\cdot x_j}{a_{P(i),i}};$$

end

Figure 9: 高斯消去法解聯立方程式

在以上的演算法中, 用了一種新的控制結構 **for**. 這個控制結構可以用 **while** 來寫. 例如: **for** $i=1,2,\cdots,n$ **do** S_1; 可以改寫成

$i=1$;

while $i \leq n$ **do begin** S_1; $i=i+1$ **end**;

同理, **for** $i = n, n-1, \cdots, 1$ **do** S_1 **end** 可以改寫成

$i = n$;

while $i \geq 1$ **do begin** S_1; $i = i - 1$ **end**;

7 搜尋的方法

以上的演算方法均屬數值計算, 其主要目的是要計算某個變數的值. 在計算機的應用上, 除了計算數值之外, 還有一些重要的應用其所強調的並非精確的數值計算. 其中最常見的就是搜尋和排序.

假設有 n 個學生的資料, 每個學生的資料包含學號, 姓名, 各科成績, 等. 每一項資料都用一個 array 來儲存, 每一個 array 中 第 i 個位址所儲存的資料是屬於同一個學生的. 假設學號是儲存在 array A,

var A: **array** $[1..n]$ **of integer**;

用學號來查詢學生的成績的簡單方法爲在 array A 中一一尋找. 這個方法稱爲 sequentail search, 其演算法如圖 10.

輸入: 學號 x.

輸出: i, 使得 $A[i] = x$ ($i = n+1$ 表示沒有學生的學號爲 x).

方法:

$i = 0$;

repeat $i = i + 1$ **until** ($A[i] = x$ **or** $i = n$);

if $A[i] \neq x$ **then** $x = n + 1$;

Figure 10: Sequentail search

在以上的方法中, 每個迴圈都要判斷 $A[i] = x$ 和 $i = n$, 前者是要判定我們所要的資料是否已找到了, 後者是要判定所有的資料是否已找遍了. 有一個很好的方法來加快 sequentail search 的速度. 首先要將 array 的儲存空間放大一個儲存格.

var A: **array** $[1..n+1]$ **of integer**;

然後在搜尋之前先將 x 的值放在 $A[n+1]$. 這樣做之後, 必然會有一個 i, 使得 $A[i]=x$, 因此, 在搜尋的迴圈中不需要判斷是否會超過了 array A 的範圍了. 這個方法稱爲 sequentail search with sentinel, 其演算法如圖 11.

輸入: 學號 x.

輸出: i, 使得 $A[i]=x$ ($i=n+1$ 表示沒有學生的學號爲 x).

方法:

$i=0$;

$A[n+1]=x$;

repeat $i=i+1$ **until** $A[i]=x$;

Figure 11: sequentail search with sentinel

如果所建立的學生資料是按照學號由大到小排列的, 還可以設計更好的一演算法來找到我們所要的資料. 圖 12 介紹 binary search. Binary search 先比較中間的資料, 如果我們要找的資料比較小, 則下次在上半部找, 否則就在下半部找. 如此反覆進行, 直到我們的資料找到了, 或是所要找的範圍已經沒有資料了爲止. 在尋找的過程中, 我們將尋找的範圍用兩個變數 i 和 j 來表示. 一開始, $i=1$, $j=n$, 表示我們要找的範圍是從 $A[1]$ 到 $A[n]$. 在尋找的過程, $i \leq j$ 表示還有資料沒有找完, 因此, 要尋找到 $i>j$ 才能結束.

假設有 n 個資料. 在最差的情況下, sequential search 要做 n 次比較. Binary search 在每次比較之後, 可以將所要尋找的資料個數減半, 因此, 若有 n 個資料, 最多只需要 $\lfloor \log_2 n \rfloor + 1$ 次的比較就可以結束, 當 n 很大的時候, 比 sequential search 要快多了.

8 排序的方法

爲了簡單起見, 假設要排序的是 n 個整數 $a_1, a_2, \ldots, a_n$, 並且假設這 n 個數是儲存在一個一維向量 a 之內, 即 $a[i]$ 是存 a_i 的地方. 排序的方法有很多,

輸入: 學號 x.

輸出: i, 使得 $A[i] = x$ ($i = n + 1$ 表示沒有學生的學號爲 x).

方法:

$i = 1$; $A[n + 1] = x$;

repeat

  $k = (i + j)$ **div** 2;
  if $x > A[k]$ **then** $i = k + 1$ **else** $j = k - 1$;

until ($A[k] = x$ **or** $i > j$);

if $A[k] = x$ **then** $i = k$ **else** $i = n + 1$;

Figure 12: binary search

我們將學習三種基本的排序的方法: 插入法, 選取法以及快速法.

8.1 插入排序法 (Insertion Sort)

插入法所採取的方式就像我們玩撲克牌一樣, 將發到的新牌放入手中已排好的牌中的適當位置上.

在此法中, 一維向量 $a[1, \cdots, n]$ 可以分成兩個部份, $a[1, \cdots, i]$ 與 $a[i + 1, \cdots, n]$, 其中 $a[1, \cdots, i]$ 代表已排好的部份, 而 $a[i + 1, \cdots, n]$ 爲尚待排序的部份. 在一開始時 $i = 1$, 即只有 $a[1]$ 這個部份已排好. (因爲 $a[1, \cdots, 1]$ 只包含一個元素, 因此必然是已排好了.) 而 $a[2, \cdots, n]$ 則爲尚待排序的部份.

插入法首先將尚待排序之第一個元素插入 $a[1, \cdots, i]$ 中的適當位置. 如此做 $n - 1$ 次之後, 就可以將整個序列排序完成. 其演算法如圖 13.

在上述的演算法中, 要將 x 插入 $a_1, a_2, \cdots, a_i$ 中時用了一個很重要的技巧. 現在將此技巧説明如下: 首先 x 要和 $a_i, a_{i-1}, \cdots$ 等元素比較. 假設是由小到大排列, 而 $a_i, a_{i-1}, \cdots, a_{i-k}$ 均大於 x, 則 $a_i, a_{i-1}, \cdots, a_{i-k}$ 各要往後移一個位子. 直到某個數 a_j 比 x 小時爲止. 這個動作是在第二層的 **while** 中

輸入: $n, x_1, x_2, \cdots x_n$.

輸出: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$.

方法:

for $i = 2$ **to** n **do begin**

$x = a_i$; $a_0 = x$; $j = i - 1$;

while $x < a_j$ **do begin**

$a_{j+1} = a_j$; $j = j - 1$;

end;

$a_{j+1} = x$;

end

Figure 13: Insertion Sort

執行的. 在執行這個 **while** 迴圈之前, 可以先將 $a[1, \cdots, n]$ 這個向量往前加一元素及 $a[0]$, 而讓 $a[0]$ 等於 x 之值. 如此做就可以不必在 **while** 迴圈中測試 j 的值是否會小於 1 了, 因爲在 $j = 0$ 時 $a[j] = x$ 必會使 $x < a[j]$ 不成立. 這種技巧可以使內層的 **while** 迴圈少掉一個比較, 而內層的迴圈是被執行最多次的迴圈, 如此一來, 就可以節省執行的時間了.

當然, 想要眞正減少元素間比較的次數, 而使程式加快, 就非要改演算法不可. 因爲 $a[1, \cdots, i]$ 是一段已經排序好的部份, 因此可以用二元搜尋法而找到 x 應該插入的位置. 二元搜尋法就是不要一個一個的比較, 而直接比較中間那個元素, 若中間的元素比 x 小, 則知道 x 應在前半段. 反之, 若中間的元素比 x 大, 則 x 應在後半段. 如此反覆做, 就像 binary search 一樣.

如此改進之後, 元素和元素之間所需要的比較次數就會大幅減少, 此點當 n 是很大的時候就特別顯著. 但是此法所需移動元素的次數仍然是和原來的程式一樣多.

8.2 選擇排序法 (Selection Sort)

選擇法可能是一種最直覺的方法.它首先從 $a_1, a_2, \cdots, a_n$ 中選則最小的元素,

輸入: $n, x_1, x_2, \cdots x_n$.

輸出: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$.

方法:

for $i = 2$ **to** n **do begin**

$x = a_i$; $l = 1$; $r = i - 1$;
while $l \leq r$ **do begin**
$m = (l + r)/2$;
if $x < a_m$ **then** $r = m - 1$ **else** $l = m + 1$ **end**;
for $j = i - 1, i - 2, \ldots, l$ **do** $a_{j+1} = a_j$;
$a_l = x$;

end

Figure 14: Insertion Sort using Binary Search

然後將此元素與 a_1 交換, 因此 $a[1, \cdots, 1]$ 是已經排好的部份, 而 $a[2, \cdots, n]$ 則是尚待排序的部份.然後再從$a_2, \cdots, a_n$ 中選擇最小的元素與 a_2 交換, 如此 $a[1, \cdots 2]$ 即爲已排好的部份, 而$a[3, \cdots, n]$ 則尚待排序. 如此反覆 $n - 1$ 次之後, 整個數列 $a_1, a_2, \cdots, a_n$ 就排序完成了. 整個演算法如圖 15.

8.3 快速排序法 (Quick Sort)

快速法所根據的原理是先任選一元素 x, 通常這個元素是存放在中間的元素, 即 $x = a[n/2]$. 然後將 $a_1, a_2 \cdots a_n$ 分成三個部份: (1) $[a_1, \cdots, a_j]$, (2) $[a_{j+1}, \cdots, a_i]$, (3) $[a_{i+1} \cdots a_n]$, 其中第一部份元素之值均小於或等於 x, 第二部份元素之值均等於 x, 而第三部份元素之值均大於或等於 x. 因爲我們要從小到大排列, 因此下一步驟只要反覆排列 $[a_1 \cdots a_j]$ 及 $[a_{i+1} \cdots a_n]$ 即可. 當然若其中任一部分只包含一個元素, 那麼就可以不用再排了.

當 x 選定之後要如何分成三個部份 $[a_1, \cdots, a_j]$, $[a_{i+1}, \cdots, a_i]$ 以及 $[a_{i+1}, \cdots, a_n]$ 使得第一部份之元素小於或等於 x, 而第二個部份之元素等於 x, 而第三部份之元素大於或等於 x 呢? 其方法如下: 從第一個元素開

輸入: $n, x_1, x_2, \cdots x_n$.

輸出: $x_{(1)}, x_{(2)}, \cdots x_{(n)}$.

方法:

for $i = 1$ **to** $n-1$ **do begin**

$k = i$; $x = a_i$;

for $j = i+1$ **to** n **do begin**

if $a_j < x$ **then** $k = j$; $x = a_j$ **end**;

end;

$a_k = a_i$; $a_i = x$;

end

Figure 15: Selection Sort

始, 比較其是否小於 x. 若是, 則這個元素已在其應屬的部位, 若否, 則從最後面的一個元素開始, 比較其是否大於 x. 若是, 則這個元素已在其應在的部位, 若否, 則找到了兩個元素, 一個大於或等於 x, 一個小於或等於 x, 而各不在其應在的部位. 我們即將這兩個元素交換. 交換之後, 這兩個元素就被放在它們應在的部位了. 如此反覆進行, 直到, 全部元素均被看完時爲止, 就可將原來之元素 $a_1 a_2 \cdots a_n$ 分成三個部份, 其前段小於或等於 x, 後段大於或等於 x, 而中段等於 x. 整個演算法如圖 16.

上述之演算法要用到自己呼叫自己的特性, 這種呼叫方式在 BASIC 語言中是不允許的. 但是若將需要排序的部份, 存在一個 stack 中. 首先將 $[1, n]$ 存入, 表示要排序 $a_1 \cdots a_n$. 每次檢查 stack 是否有要排序的元素. 若有, 即將其從 stack 拿出來排序 (即分成三段). 然後再檢查前後段之元素是否多於一個. 若是, 則此段還需要繼續做排序的動作, 將其上下界存到 stack. 就可以避免反覆呼叫自己的動作了其演算法如圖 17.

輸入: $n, x_1, x_2, \cdots x_n$.

輸出: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$.

方法:

procedure qsort(l, r: index);

$i = l$; $j = r$;

$x = a_{(l+r)/2}$;

repeat

  while $a_i < x$ **do** $i = i + 1$ **end**;
  while $x < a_j$ **do** $j = j + 1$ **end**;
  if $i \leq j$ **then begin**
    $w = a_i$; $a_i = a_j$; $a_j = w$;
    $i = i + 1$; $j = j - 1$;
  end;

until $i > j$;

if $l < j$ **then** qsort(l, j);

if $i < r$ **then** qsort(i, r);

end;

begin

qsort(1,n);

end.

Figure 16: Quick Sort

輸入: $n,\ x_1, x_2, \cdots x_n$.

輸出: $x_{(1)} \leq x_{(2)} \leq \cdots \leq x_{(n)}$.

方法:

$s = 1$; $stack[1] = [1, n]$;

repeat

  $[l, r] = stack[1]$; $s = s - 1$;

  repeat

    $i = l$; $j = r$; $x = a_{(l+r)/2}$;

    repeat

      while $a_i < x$ **do** $i = i + 1$ **end**;

      while $x < a_j$ **do** $j = j + 1$ **end**;

      if $i \leq j$ **then**

        $w = a_i$; $a_i = a_j$; $a_j = w$;

        $i = i + 1$; $j = j - 1$;

      end

    until $i > j$;

    if $i < r$ **then begin**

      $s + 1$; $stack[s] = [i, r]$;

    end;

    $r = j$;

  until $l \geq r$;

until $s = 0$;

Figure 17: Nonrecursive Quick Sort