Scilab 簡介 3

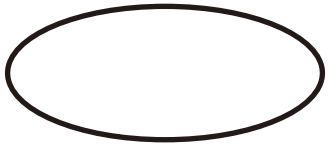
2011/10/06

Chih-Han Lin 林致翰

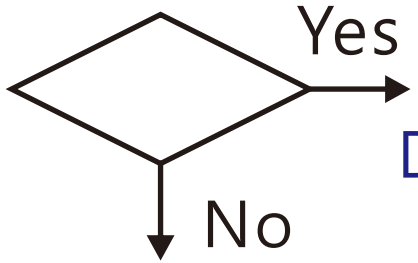
clin@ltl.iams.sinica.edu.tw
or r99245002@ntu.edu.tw

流程圖（flow chart）

A flowchart is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. This diagrammatic representation can give a step-by-step solution to a given problem. (from wikipedia, “Flowchart”)



Initial condition
Start / End

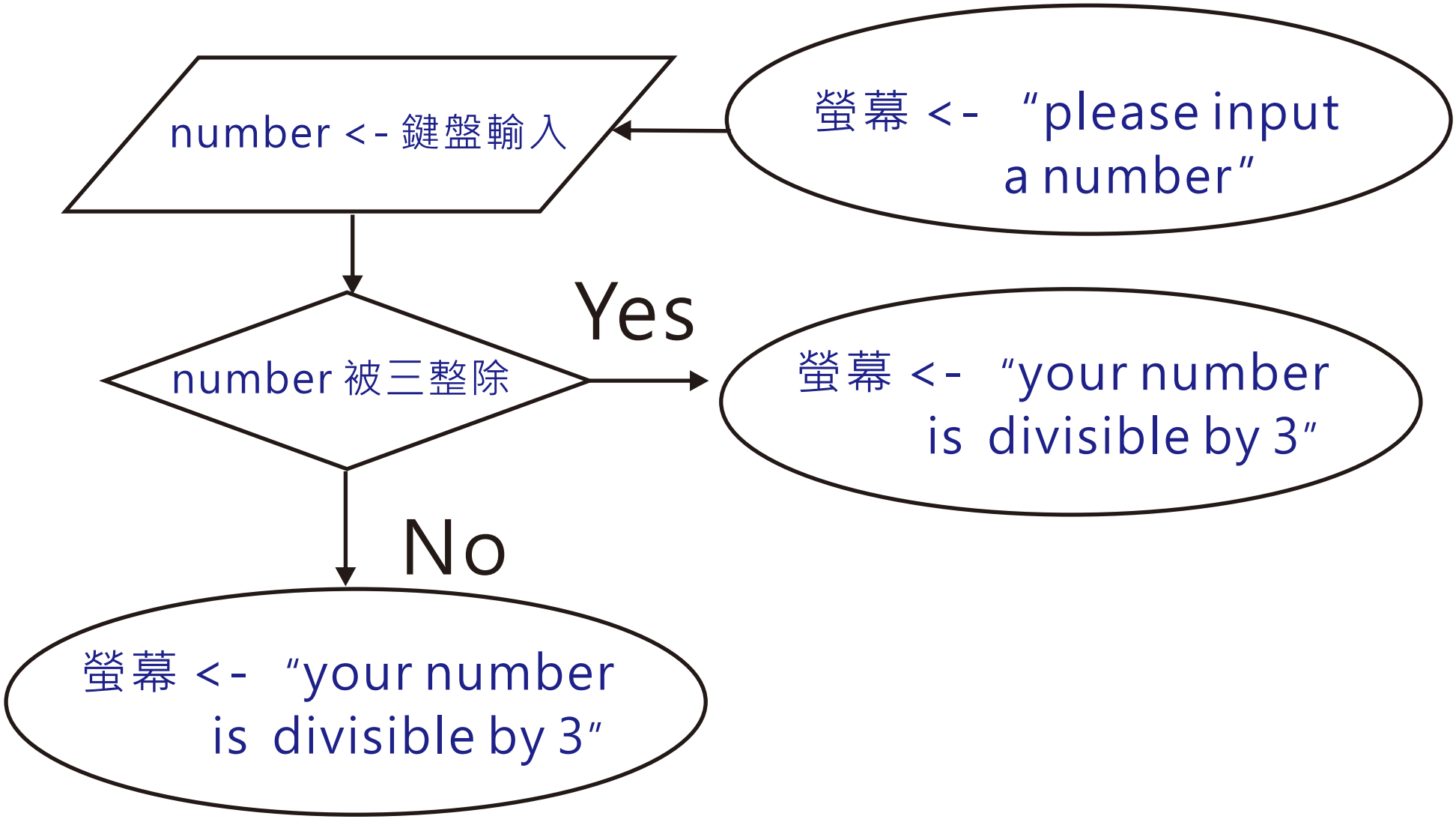


Decision

Statment

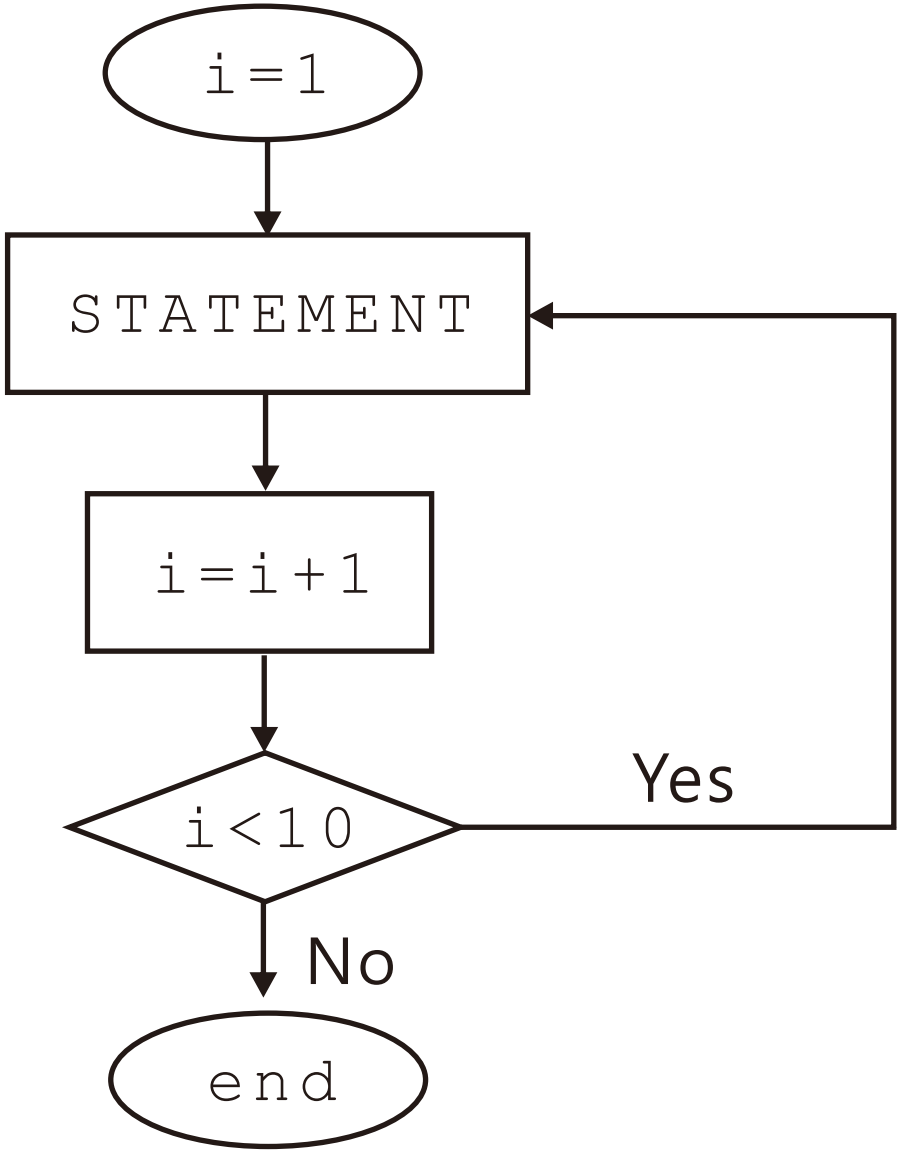
Order / data flow

ex. 寫一個可以判定使用者數值是否為 3 的變數的程式



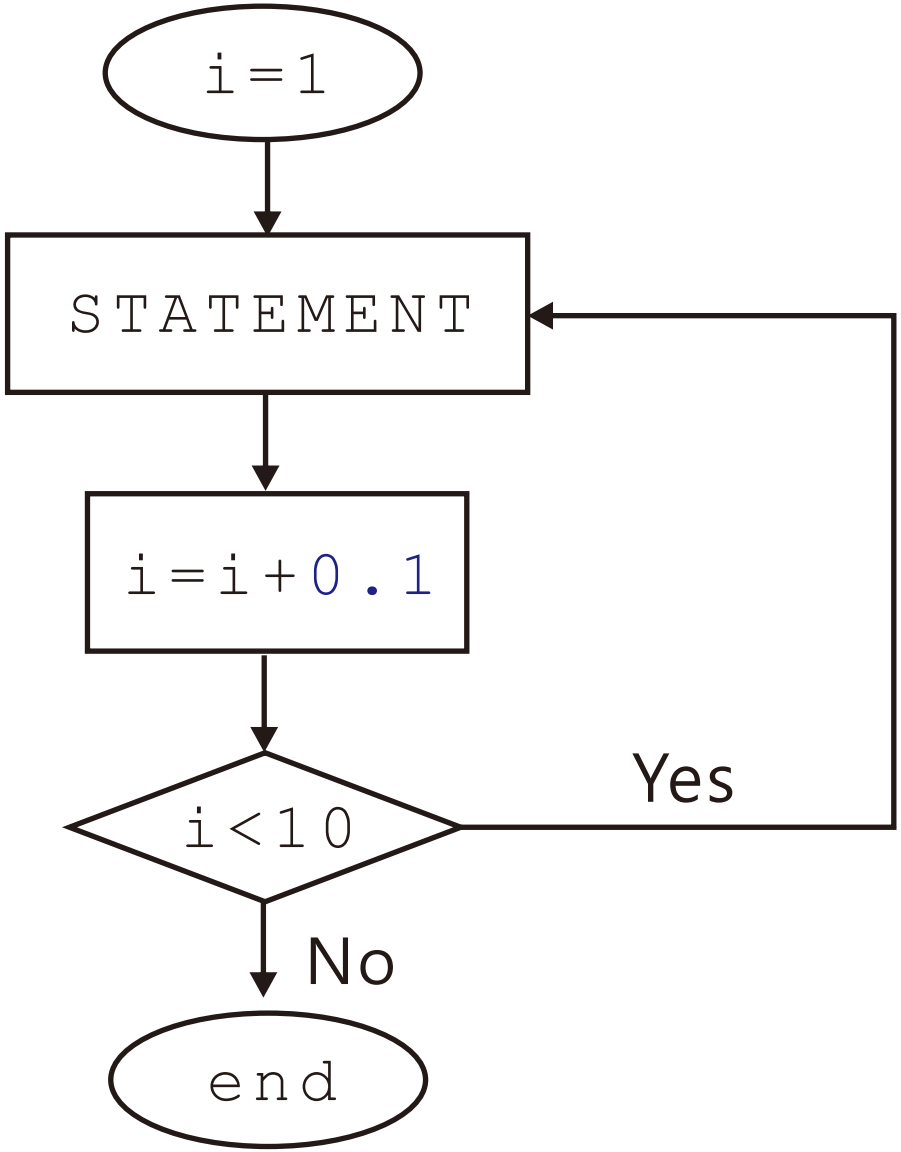
for 迴圈流程圖

```
for i=1:10
STATEMENT
end
```



for 迴圈流程圖

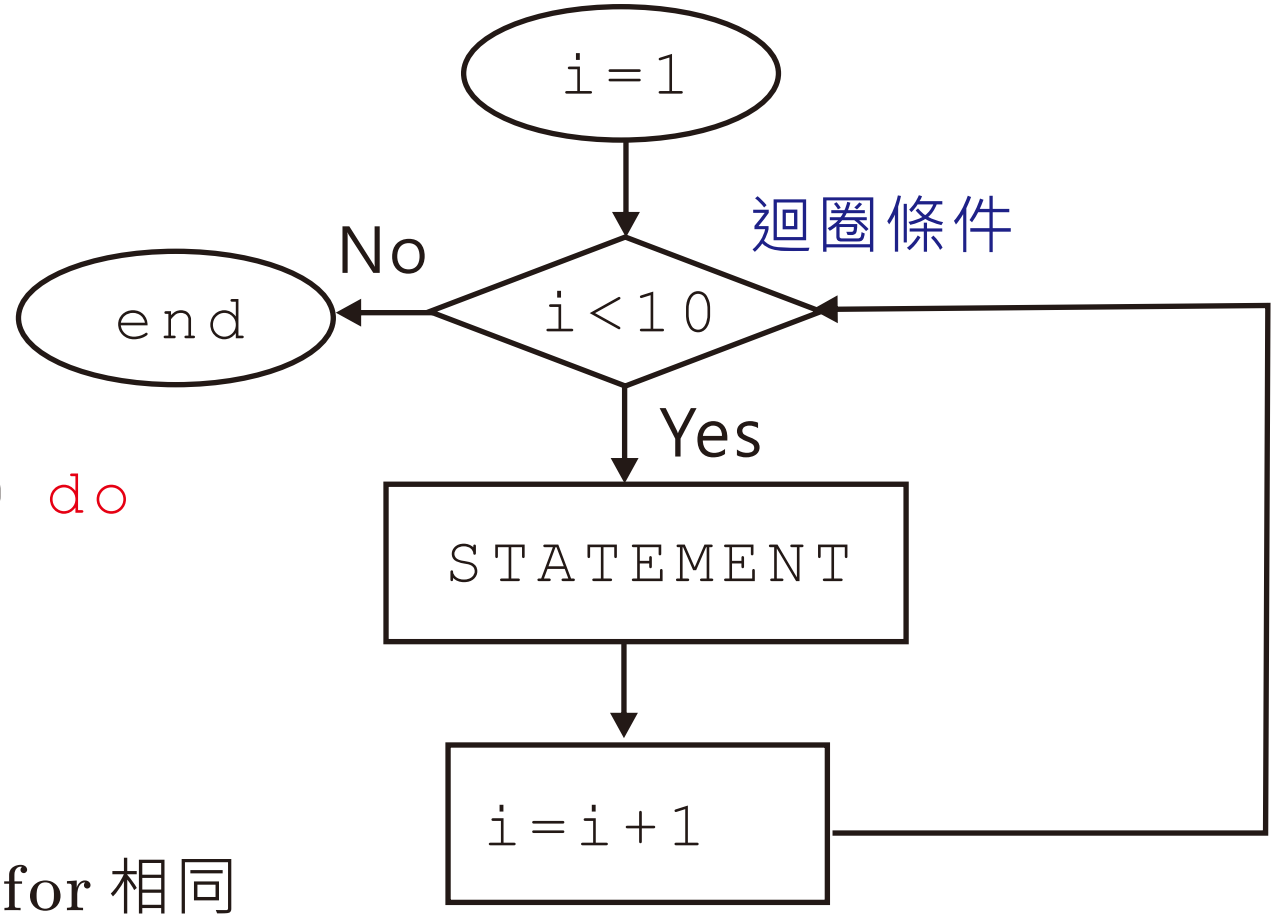
```
for i=1:0.1:10
STATEMENT
end
```



while 迴圈

迴圈條件

```
i=1;
while i<10 do
STATEMENT
i=i+1
end
```



某種程度上效果與 for 相同

根據迴圈條件很容易寫成無法停止的無窮迴圈

ex. 計算一個公比 0.9，首項 1 的無窮等差級數和，必須累加幾項才能讓誤差小於 1%（ $1/(1-0.9)=10$ 為理論值）

```
i=0;
total=0;
while (10-total)>0.1 do
    total=total+0.9^i;
    i=i+1;
end
printf('sum of %d terms, total=%f',i,total);
```

ex. 計算一個公比 0.9，首項 1 的無窮等差級數和，必須累加幾項才能讓誤差小於 1%（ 1/(1−0.9)=10 為理論值 ）(使用 for)

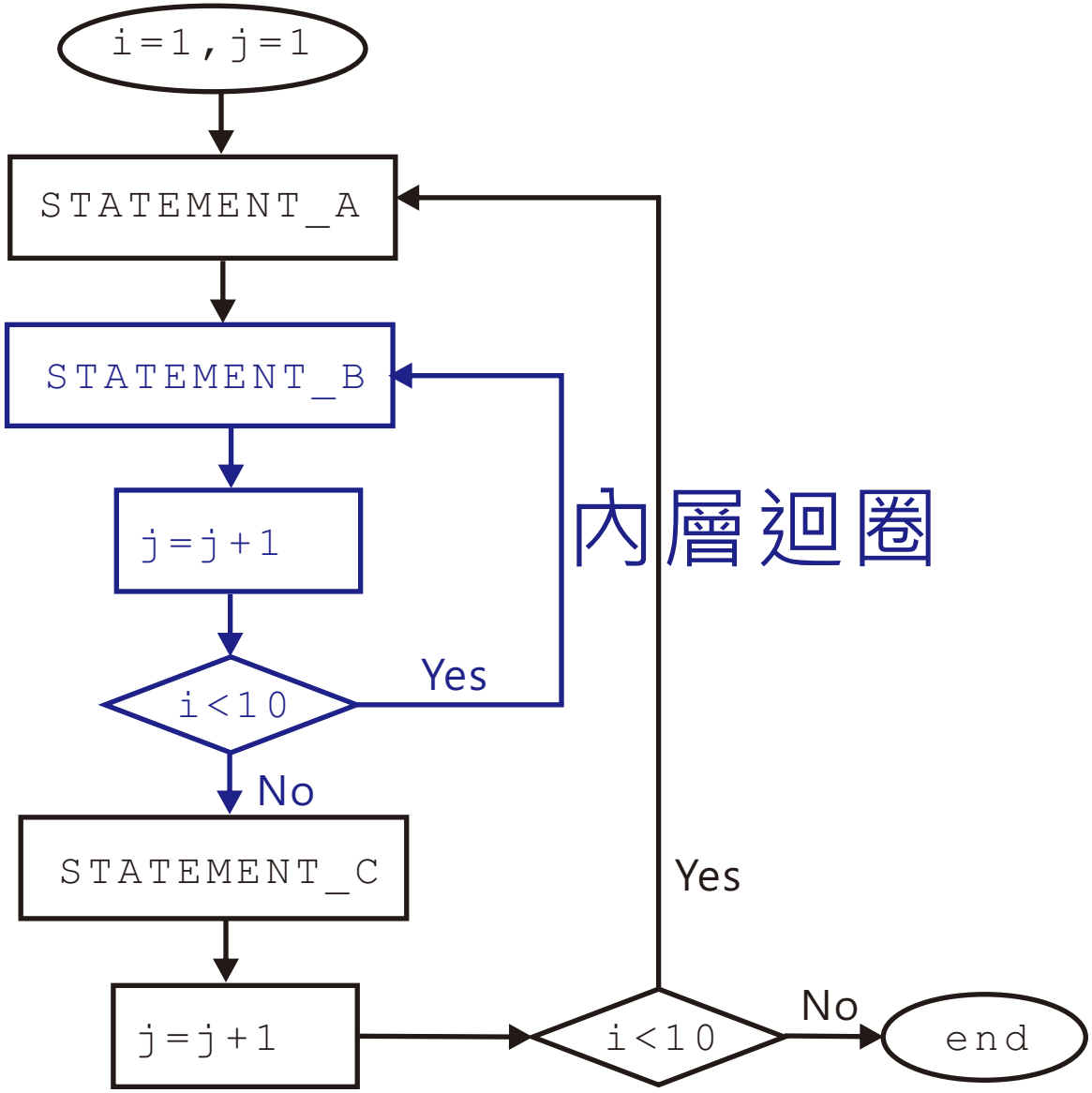
```
total=0;
for i=0:1000
    total=total+0.9^i;
    if (10-total)<0.01 then
        break
    end
end
printf('sum of %d terms, total=%f',i+1,total);
```

Q : 同樣功能的程式碼用 while 迴圈來作比較快還是用 if +break 來作比較快？

以這裡的範例為例，用 if 來寫比 while 快了約 20%

巢狀迴圈

```
for i=1:10,
    STATEMENT_A;
     for j=1:10,
       STATEMENT_B;
     end
    STATEMENT_C;
end
```



ex. 在螢幕上列印 4x4 乘法表

```
for i=1:4,
    for j=1:4,
        printf('%d x %d = %d  ',i,j,i*j);
    end
    printf('\n');
end
```

螢幕上顯示結果：

```
1 x 1 = 1  1 x 2 = 2  1 x 3 = 3  1 x 4 = 4
2 x 1 = 2  2 x 2 = 4  2 x 3 = 6  2 x 4 = 8
3 x 1 = 3  3 x 2 = 6  3 x 3 = 9  3 x 4 = 12
4 x 1 = 4  4 x 2 = 8  4 x 3 = 12  4 x 4 = 16
```

```
for i=1:4,
    for j=1:4,
        printf('%d x %d = %d  ',j,i,i*j);
    end
    printf('\n');
end
```

i,j 互換

螢幕上顯示結果：

```
1 x 1 = 1   2 x 1 = 2   3 x 1 = 3   4 x 1 = 4
1 x 2 = 2   2 x 2 = 4   3 x 2 = 6   4 x 2 = 8
1 x 3 = 3   2 x 3 = 6   3 x 3 = 9   4 x 3 = 12
1 x 4 = 4   2 x 4 = 8   3 x 4 = 12   4 x 4 = 16
```

函數

Scilab 有許多內建函數，根據你給定的參數輸出你要的結果或者進行某樣行為： ex. str=string(12) 將接收括號的數值 12 轉變為字串型態 '12' 並將輸出結果存至變數 str 中。

Scilab 也允許使用者建立自定義的函數
合理地撰寫函數可以

a) 增加程式的可讀性，減少 dirty work
b) 隔離不同作用的變數，減少出錯的機會 (local variable)
c) 程式模組化

```
function [a1, a2,... ]=FunctionName(x1, x2,..)
statement
endfunction
```

ex. 撰寫一個函數，將 x, y 座標轉成極座標

```
function [mag, theta]=Polc(x,y)
mag=sqrt(x^2+y^2);
if (x>0)|(x<0) then
theta=atan(y/x)
elseif y>=0 then
    theta=%pi/2;
elseif y<0
    theta=3*%pi/2;
end
endfunction
```

函數輸出只能放在等號左邊

在 Scinote 執行上頁寫好的函數，以後在 console 或者其他 scinote 寫好的批次檔便可以調用該函數

```
-->[mag1, theta1]=Polc(1,1)

 theta1  =  0.7853982
 mag1  = 1.4142136

-->[mag1, theta1]=Polc(0,3)

 theta1  =  1.5707963
 mag1  = 3.
```

Global 變數與 Local 變數

呼叫一個函數，該函數中所宣告的變數如果沒有特別聲明為 global 變數的話，那麼預設上該函數中宣告的變數屬於 local 變數。該函數中的 local 變數只有該函數可識別、調用，在該函數被呼叫的當下被創造出來，當該函數結束呼叫，回傳輸出結果給主程式時，則 local 變數會自動被清除（ clear ）

同一個函數被多次呼叫時，local 變數不會被共用，換言之每呼叫一次便生成一次 local 變數供該被呼叫函數專用。

global 變數則代表主程式中宣告（不在 function 的 block 內 的變數），只要不在 console 中執行 clear 便一直存在、占用記憶體空間的變數資料

函數名&參數

```
function foo()
    a=100;
    disp(a);
endfunction
```

宣告一個函數 foo，在函數中宣告 (local) 變數 a 為 100，將 a 印至螢幕上

```
-->deff('foo()',['a=100'; 'disp(a)'])
```

deff 為 的等價寫法，當 function 比較簡單時可以用 deff 直接寫成一行在 console 中宣告

```
-->foo

100.

--> a=50; foo

100
```

在 console 中宣告的 a 跟 foo 中的 a 毫無關係

foo 中沒定義的變數 a，默認為主程式中的 a

```
--> clear;
--> deff('foo()',['disp(a)'])

-->foo

   !--error 4 未定義變數:a

 --> a=10; foo

     10.
```

```
--> clear;
--> deff('foo()',['global a','a=100','disp(a)'])
--> deff('goo()',['global a','disp(a)'])

-->a=20; foo

    100.

--> goo

    100.
--> global a; a=59; goo

    59

--> foo

    100.
```

函數中被宣告為 global 的變數可以被其他函數使用，比如 goo 中引用的 a 值與 foo 相同，但跟 console 的不同

函數之中仍然可以修改在函數中宣告為 global 之變數

遞迴（recursion）

與數學歸納法的精神相同，設定終止條件，遞迴規則並在宣告函數過程中出現呼叫被宣告函數的情況即稱之為遞迴（recursive function call）。

通常使用遞迴 可以將程式碼寫得異常精簡，缺點則在於每次出現 recursive function call 都會製造 local 變數堆疊佔用記憶體空間，一不注意便可能將記憶體塞滿而出錯，另一方面大量創造消滅變數的過程也會耗用大量計算資源、降低程式執行的效率。

一般的應用中不建議使用遞迴寫法。

ex. 撰寫一個遞迴函數 ssu(n)，可以計算 1+2+3+...+n 之值

```
function total=ssu(n)

    if n==1 then
        total=1;
    else
        total=n+ssu(n-1);
    end
endfunction
```

設定遞迴起始點

設定遞迴規則

$ssu(n)=1+2+3+...+n=(1+2+3+...+n-1)+n=ssu(n-1)+n,$
$ssu(1)=1.$

```
--> ssu(100)

 5050.
```

ex. 撰寫一個非遞迴函數 ssu(n)，可以計算 1+2+3+...+n 之值

ssu2(n) 使用 for 迴圈寫法

```
function total=ssu2(n)
    temp=0;
    for i=1:n
        temp=temp+i;
    end
    total=temp
endfunction
```

ssu3(n) 使用陣列相關指令

```
function total=ssu3(n)
    total=sum(1:n);
endfunction
```

Speed Test (timer 指令)

```
tic();
STATEMENT
q=toc();
```

tic 設定計時起始點，toc 設定終點
一般將待測速的程式碼放於兩者之間

```
roundtrip=1000;
tic();
for i=1:roundtrip,
STATEMENT
end
q=toc()/roundtrip;
```

很多時候函數執行一次所花得時間非常少，我們通常會利用 for 迴圈讓它執行多次以求出合理的平均耗時

測試前一個範例中利用不同方法寫成的求和函數 ssu, ssu2, 與 ssu3 的執行速度

```
roundtrip=10000;
tic();
for i=1:roundtrip
    ssu(100)
end
q=toc();
printf('consume time of ssu = %f\n',q);
tic();
for i=1:roundtrip
    ssu2(100)
end
q=toc();
printf('consume time of ssu2 = %f\n',q);
tic();
for i=1:roundtrip
    ssu3(100)
end
q=toc();
printf('consume time of ssu3 = %f\n',q);
```

測速結果：

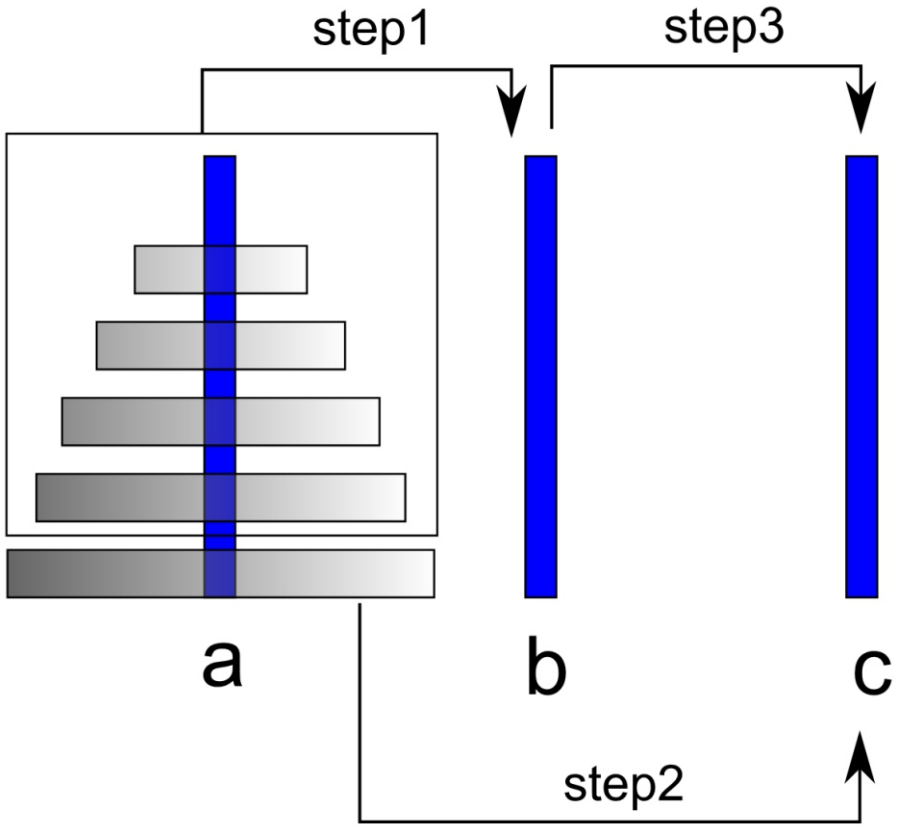
```
consume time of ssu = 18.521000
consume time of ssu2 = 1.912000
consume time of ssu3 = 0.260000
```

遞迴寫法最慢（事實上當 n = 1074 以上時程式便無法正常執行）
for 迴圈的寫法也比直接使用陣列指令慢上許多

在 Scilab 中如果能寫成陣列來處理儘量使用陣列來作
程式的處理效率會比較好

遞迴經典範例（河內塔）

一次只能移動一片
小的塔片只能放在大的塔片上，要把 a 柱上的 n 層塔轉移到 c 柱上，可以利用 b 柱暫存



```
function hanoi(n,a,b,c)
    if n==1 then
        printf('move sheet %d from %s to %s \n',n,a,c);
    else
        hanoi(n-1,a,c,b);
        printf('move sheet %d from %s to %s \n',n,a,c);
        hanoi(n-1,b,a,c);
    end
endfunction
```

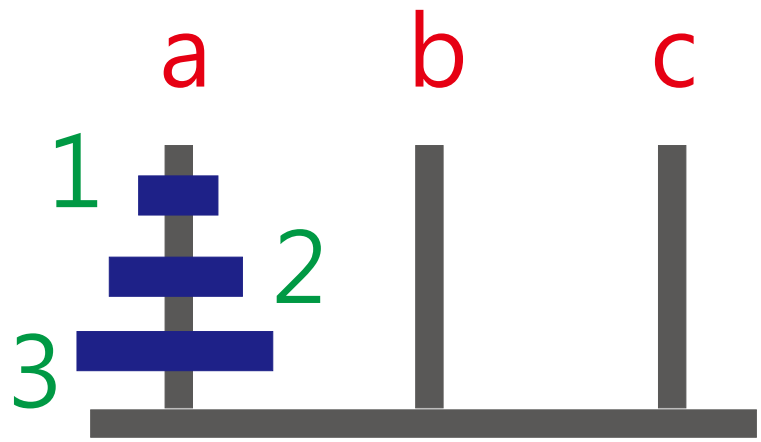
step1: 把上面 n-1 層從 a 移到 b ：呼叫 hanoi(n-1, a, c, b)

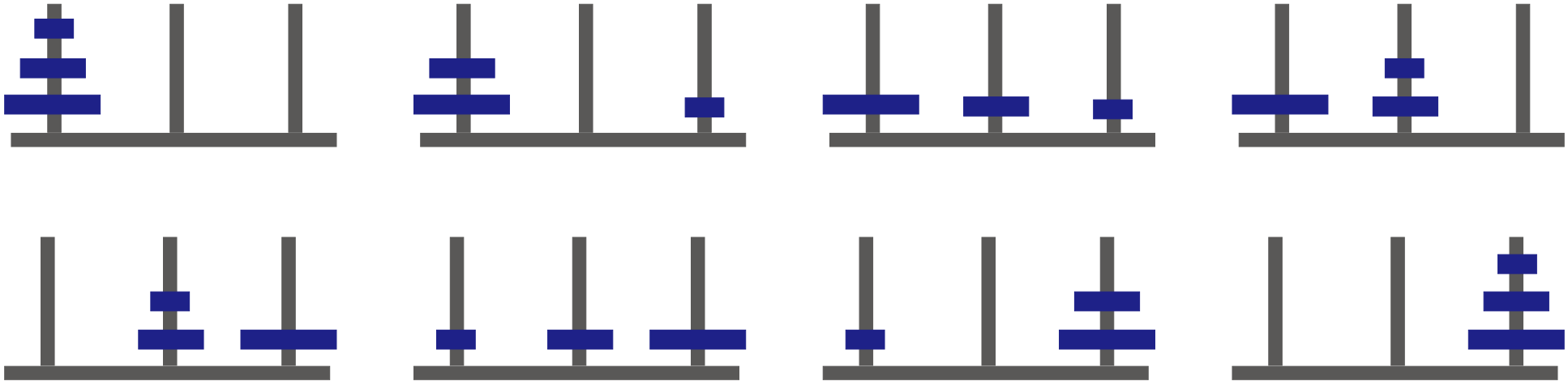
step2: 把第 n 層從 a 移到 c

step3: 把上面 n-1 層從 b 移到 c ：呼叫 hanoi(n-1, b, a, c)

```
--> hanoi(3,'a','b','c')

move sheet 1 from a to c
move sheet 2 from a to b
move sheet 1 from c to b
move sheet 3 from a to c
move sheet 1 from b to a
move sheet 2 from b to c
move sheet 1 from a to c
```





實作內容

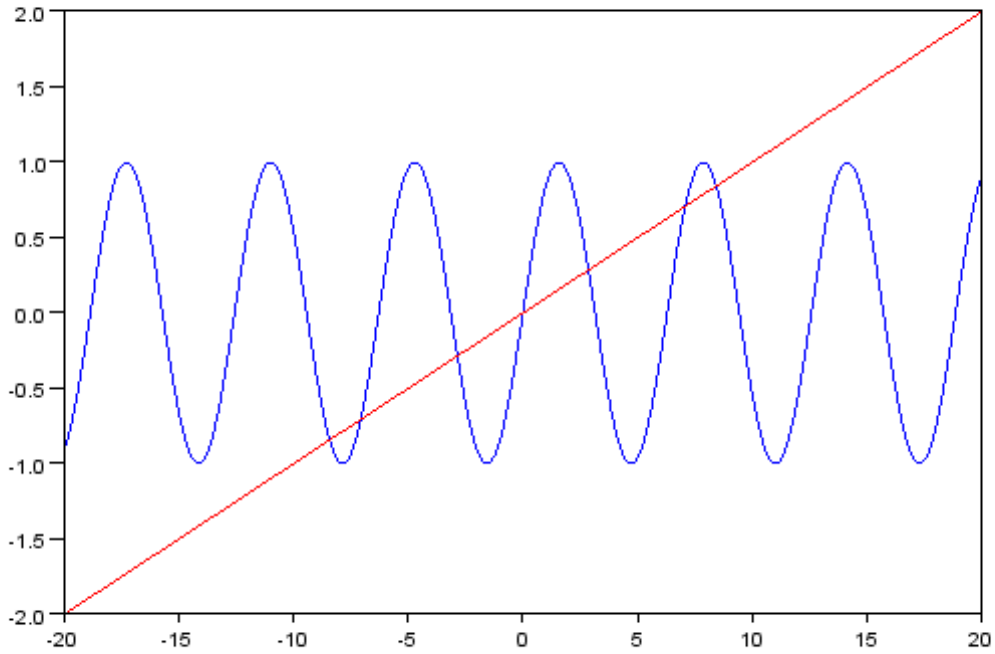
請寫一個函數 sarray(n) ，其功能為將 1,2,3,..., n^2 個數字以 S 型由小到大排到 n x n 的表格中，印至螢幕上
你必須同時畫出程式的流程圖 (flow chart)

ex.

```
--> sarry(4)

 1.    2.    3.    4.
 8.    7.    6.    5.
 9.   10.   11.   12.
16.   15.   14.   13.
```

畫出 -20<x<20 時，y=0.1*x 與 y=sin(x) 的曲線，
判斷 0.1*x= sin(x) 這個方程式是應該會有幾個解（x_n)
接著撰寫 while 迴圈來求出近似數值解 x_n 的值使得
0.1*x_n-sin(x_n) 小於 0.0001
同樣你必須畫出程式的流程圖（flow chart）



行事曆上事先公佈的實作題目

1. 以非遞迴 (recursion) 的寫法撰寫一個函數 `fibonacci1`，呼叫此函數 `fibonacci1(n)` 可以獲得第 `n` 項斐波那契數。
2. 修改 `fibonacci1` 使得在命令列調用此函數可以在螢幕上印出前 `n` 位斐波那契數，比如說 `fibonacci(7)` 可以在螢幕上顯示前 7 位斐波那契數 0112358 。
3. 使用遞迴法 (recursion) 來撰寫一個計算 `fibonacci` 的函數，呼叫此函數 `fibonacci2(n)` 可以獲得第 `n` 項斐波那契數。
4. 分別以 `fibonacci1` 與 `fibonacci2` 計算第 30 項斐波那契數。兩支程式的計算速度有何差別？ (你可以同時開啓 variable browser 來觀察)