資料結構簡介

Chun-Jung Lin

國立勤益科技大學資訊工程系
Department of computer science and information engineering

OUTLINE

- 資料結構的定義
- 資料結構影響程式執行效率
- 演算法的定義
- 程式執行效率分析
- 評估程式的複雜度

何謂資料結構 (1/3)

- 程式設計就是資料結構與演算法，撰寫程式前先規劃所需的資料結構，資料結構就會影響程式的演算法，利用演算法操作資料結構，完成預訂所需要達成的功能。

何謂資料結構 (2/3)

- 遞迴(Recursive)
  - 老鼠走迷宮的問題就是屬於「遞迴」結構。
- 陣列(Array)
  - 教室座位排列方式就是屬於「陣列」結構。
- 堆疊(Stack)
  - 碗盤的疊法、小朋友排積木、書本裝箱或座電梯等都具有後進先出的特性就是屬於「堆疊」結構。
- 佇列(Queue)
  - 排隊買票看球賽，先到先買的方式就是所謂的「佇列」結構。

何謂資料結構 (3/3)

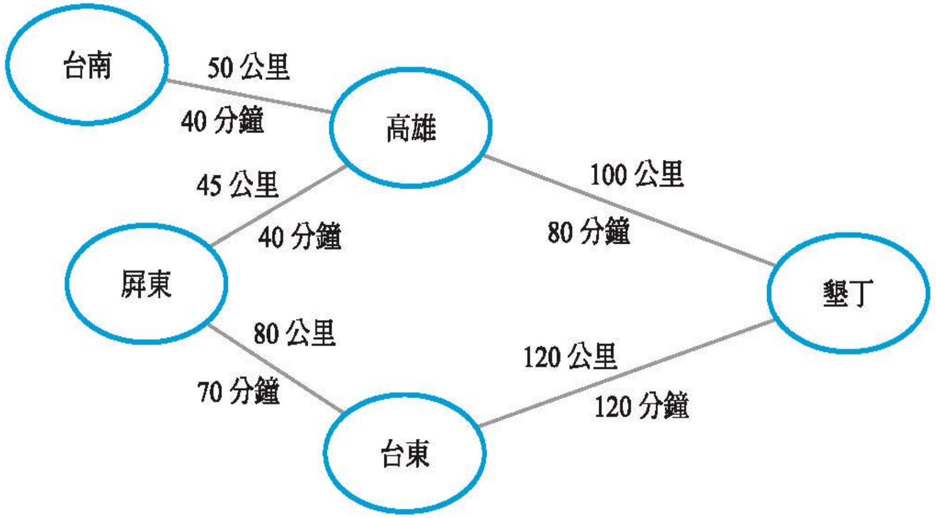
- 串列(List)
  - 高鐵火車的車箱串接方式就是屬於「串列」結構。
- 樹狀(Tree)
  - 如果球賽的賽程方式是採用淘汰制，就是一種「樹狀」結構。
- 圖形(Graph)
  - 當看完球賽要回家的行駛路線圖，則可視為所謂的「圖形」結構。
- 排序(Sort)
  - 球賽成績的結果之排名方式就是屬於「排序」結構。
- 搜尋(Search)
  - 球賽比賽之前找尋某一隊的賽程就是屬於「搜尋」結構。

資料結構的定義 (1/3)

- 資料結構(Data Structure)是電腦儲存與操作資料的方式，包含儲存的容器、新增資料、讀取資料、刪除資料與搜尋資料等。
  - 例如：要建立一個電話簿功能的程式，就要先思考適合的資料結構，該結構要能夠新增電話號碼與聯絡人，且能透過聯絡人資訊來搜尋電話號碼，並且能夠新增與刪除聯絡人和電話號碼，可以選用陣列(Array)、鏈結串列(Linked List)或字典(Dictionary)等資料儲存容器製作抽象資料型別。
- 日常生活中所遇到的問題，有時需要使用特定資料結構儲存資料，接著使用該結構所對應的演算法解決問題。

資料結構的定義 (2/3)

- 使用地圖搜尋最短時間內到達目的地的方式，可以建立地圖上每一個地點到另一個地點的距離與移動時間，儲存在圖形資料結構內，地點轉換成節點，兩個節點(地點)之間可以連通，就新增一個邊，邊上的權重就是所需距離或移動時間。
- 使用最短路徑演算法，找出距離最短或移動時間最少的路徑，就可以找出地圖上兩點之間的最短路徑或最短移動時間的規劃路線。



資料結構的定義 (3/3)

- 網路訂火車票時，訂票系統伺服器需要儲存大量的交易資料，能夠即時查詢每個列車與車廂是否有空位，能夠預定一個月以後的車票，此時需要具有能夠快速搜尋、新增與刪除功能的資料結構。

演算法的定義 (1/6)

- 為了讓電腦執行所需要的功能，必須先將這個功能轉換成演算法(Algorithm)。演算法是完成功能所需要的步驟，有了演算法才能轉換成程式。
- 資料結構也需要提供操作資料結構的演算法，不然只有儲存資料的空間，沒有操作資料的功能，就不能算是完整的資料結構。

演算法的定義 (2/6)

- 為了要讓電腦可以正確執行，演算法具有輸入(Input)、輸出(Ouput)、明確性(Definiteness)、正確性(Effectiveness)、有限性(Finiteness)之特性。
  - 輸入：演算法可能有輸入，也可能沒有輸入，如果有輸入，需要明確的說明輸入的個數與每個輸入所表示的意義。
  - 輸出：演算法至少要有一個以上的輸出，表示演算法執行後的結果。
  - 明確性：所有演算法步驟都需要明確，演算法步驟不能有兩種以上的解釋，才能依照演算法轉換成程式碼。
  - 正確性：演算法要能正確完成所需功能或解決問題，錯誤的演算法就需要修正。
  - 有限性：電腦需要在有限步驟內執行程式，若演算法無法在有限步驟內完成，演算法就無法終止，轉換的程式也無法執行完畢，無法獲得結果。

演算法的定義 (3/6)

- 演算法的表示方式，可以使用文字、虛擬碼(Pseudo Code)或流程圖(Flow Chart)進行表示。
  - 可以單純使用文字敘述解題步驟。
  - 虛擬碼使用類似程式碼的文字表示演算法，例如：利用「if」取代文字敘述的「如果」。
  - 使用流程圖表示演算法。流程圖常用於幫助程式設計者，以圖示方式寫出解決問題的步驟，若能將解題流程以流程圖表示，就可以轉換成程式語言。

演算法的定義 (4/6)

- 使用文字描述解題步驟，隨著問題的複雜度增加，可能無法清楚描述和表達。
- 虛擬碼使用類似程式碼的文字描述解題步驟，但不能夠直接執行，雖然能快速轉換成程式碼，但適合已經有程式基礎的人員，初學者對程式沒有基礎，不適合使用虛擬碼。
- 流程圖相較於文字敘述與虛擬碼，讓初學者更能掌握解題步驟，但程式規劃與設計人員需要先熟悉各種流程圖的圖形所表示的意義。使用流程圖相較於文字與虛擬碼，更適合初學者精確描述解題步驟。

流程圖圖示	意義
	程式流程的線，表示程式的處理順序。
	條件選擇，於菱形內寫入條件判斷。
	程式的敘述區塊，寫出所需完成的功能。
	開始或結束，看到此圖表示程式的開始或結束。
	程式所需的輸入與輸出。
	連接點，當流程圖過長可以使用連接點將過長流程圖切割成多個流程圖組合起來，也可以避免流程圖中線過長或交叉。

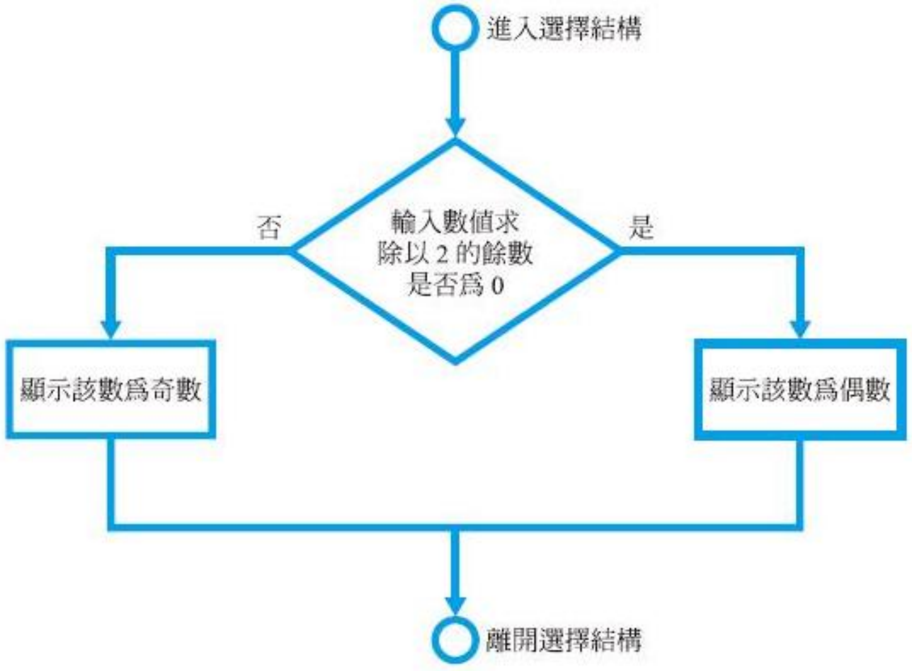
演算法的定義 (5/6)

以文字敘述表示演算法

若數字除以 2 的餘數等於 0，則數字為偶數，否則數字為奇數。

以虛擬碼表示演算法

```
if  num % 2 == 0:
    print(num是偶數)
else:
    print(num是奇數)
```



演算法的定義 (6/6)

- 演算法是解決一問題的有限步驟，而評斷演算法的優劣可利用Big-O分析之，如$O(n)$比$O(n^2)$來得佳。
- 「演算法」在韋氏辭典中定義為：「在有限步驟內解決數學問題的程序」。
- 我們可以把演算法(Algorithm)定義成：「解決問題的方法」。

步驟一：準備雞蛋、麵粉及鮮奶	步驟二：攪拌
	
步驟三：倒入模型器中	步驟四：設定烤箱加熱時間，即可完成一個蛋糕。
	

程式執行效率分析 (1/8)

- 演算法(Algorithms)
  - 解決問題(problems)的有限步驟程序。
- Example
  - 判斷數字x是否在一已排序好的數字串列s中，其演算法為：
    - 從s串列的第一個元素開始，依序的比較；
    - 直到x被發現或s串列已達盡頭；
    - 假使x被找到，則印出Yes；否則，印出No。
- 當問題很複雜時，上述敘述性的演算法就難以表達出來。演算法大都以類似的程式語言表達，並利用程式語言執行。

程式執行效率分析 (2/8)

- “他的程式寫得比我好嗎？”
  - 應該利用客觀的方法進行比較，而此客觀的方法就是複雜度分析 (complexity analysis)。
- 求出程式中每一敘述的執行次數(其中｛和｝不加以計算)，將這些執行次數加總起來。然後求出其Big-O。

程式執行效率分析 (3/8)

- 衡量程式執行效率要有共同的標準，通常以Big-O表示，Big-O是程式複雜度的上界，表示程式執行效率最差也會在此Big-O的複雜度以內，Big-O的定義如下。

O(h(n)) = { f | 存在正數 C 與正整數 N，對於每個 n >= N，使得 0 <= f(n) <= C*h(n) }

我們就可以說「h(n) 是 f(n) 的上界」，f(n) 一定不會超過 h(n)。

程式執行效率分析 (4/8)

範例 1

$2x^2+5x+3 = O(x^2)$

取 C=3，N=6，對於每個 n>=6，都滿足 $0 <= 2x^2+5x+3 <= 3x^2$

範例 2

$6x^3+1000x^2+3 = O(x^3)$

取 C=7，N=1001，對於每個 n>=1001，都滿足 $0 <= 6x^3+1000x^2+3 <= 7x^3$

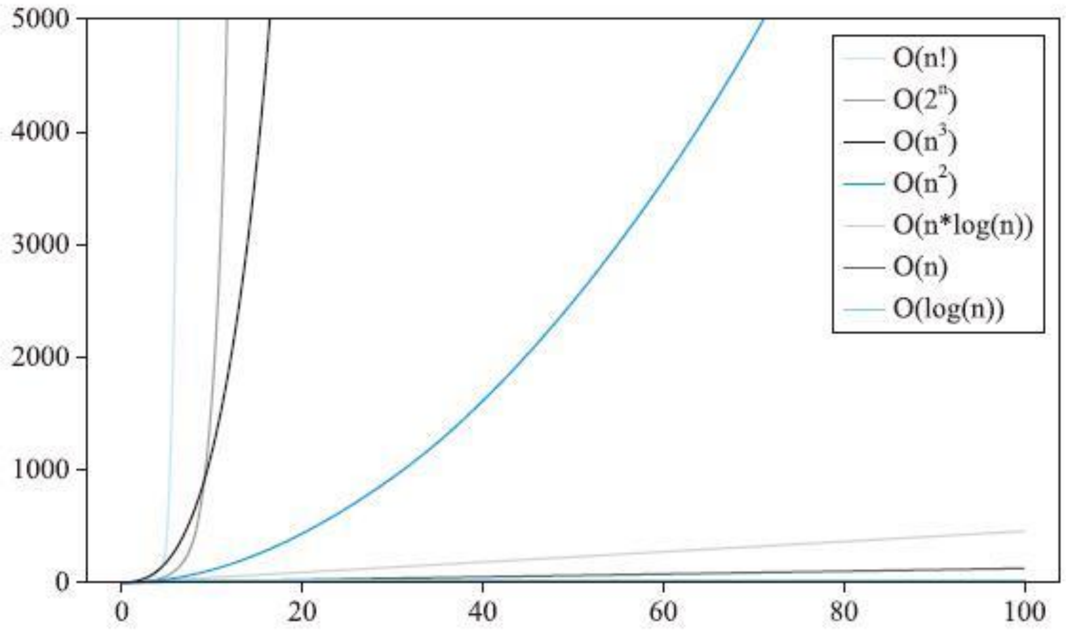
程式執行效率分析 (5/8)

- 程式複雜度越大表示越複雜，程式所需執行時間越長。程式複雜度的大小關係如下，如果執行複雜度為O(n)的程式需要花時間1秒鐘，則執行複雜度為O(n^2)的程式所需時間約為n秒鐘。

$$O(1)<O(\log(n))<O(n)<O(n*\log(n))<O(n^2)<O(n^3)<O(2^n)<O(n!)$$

程式執行效率分析 (6/8)

- 程式複雜度最小為O(1)，表示程式在常數時間內可以執行完畢，效率非常好。
- 若程式的複雜度為$O(2^n)$ 或O(n!)，表示n值每遞增1，執行演算法需要花兩倍以上的時間。
- $O(2^n)$與O(n!)成長速度很快，複雜度為$O(2^n)$與O(n!)的程式，當n值不大時，程式還能夠執行完畢，當n值夠大時，可能就無法在有限時間內執行完畢。



程式執行效率分析 (7/8)

- 程式複雜度與能處理的資料量
  - 評估程式的複雜度是以一秒鐘內可執行的資料量計算。
  - 假設演算法效率為O(n)的程式，一秒鐘大約可以完成100,000,000個資料的運算。
  - 資料量的大小與電腦中央處理器運算速度有關，隨著電腦每秒可以運算的指令數的增加，這個值會不斷成長。

程式執行效率分析 (8/8)

- 若已知演算法複雜度為O(n!)，若題目規定計算時間只有3秒鐘，則輸入的資料量n就不能超過33，n超過33就有可能逾時，就需要想效率更好的演算法才行。

演算法複雜度	n 的最大上限，這是個大概數值，會隨著電腦運算能力的進步而增加
O(n)	100000000
O(n*log(n))	4500000
O(n^2)	10000
O(n^3)	464
O(n^4)	100
O(2^n)	26
O(n!)	11

評估程式的複雜度 (1/9)

- 當寫好一個程式，就需要評估程式碼的效率。一般會以輸入資料量來計算程式的複雜度。以下為各種複雜度的範例程式。

(1) O(1)

以下程式爲比較兩數，回傳較大數值，程式的複雜度爲 O(1)。

```
if a > b:
    return a
else:
    return b
```

評估程式的複雜度 (2/9)

(2) O(log(n))

以下程式爲二分搜尋演算法。在已排序陣列中找尋某個值是否存在，每次取一半，就可以逼近要找尋的數值。執行次數約爲 O(log(n))，所以程式的複雜度約爲 O(log(n))。

```
score = [45, 59, 62, 67, 70, 78, 83, 85, 88, 92]
mid=5
left=0
right=9
while score[mid] != 59:
    print("檢查 score[", mid, "]=", score[mid],"是否等於 59")
    if left >=right:
        break
    if score[mid] > 59:
        right=mid-1
    else:
        left=mid+1;
    mid=(left+right)//2
    print("right 更新爲 ", right)
    print("left 更新爲 ", left)
```

評估程式的複雜度 (3/9)

- 搜尋的次數為log32+1=6，此處的log表示$\log_2$。
- 資料量為128個時，其搜尋的次數為log128+1，因此當資料量為n時，其執行的次數為logn+1。

陣列大小	二元搜尋	循序搜尋
128	8	128
1,024	11	1,024
1,048,576	21	1,048,576
4,294,967,296	33	4,294,967,296

評估程式的複雜度 (4/9)

(3) O(n)

如果要找出 n 個數字的最大值，使用循序搜尋就需要將每個數字看過一次，所以程式複雜度與資料量成正比，就可以說搜尋程式的複雜度爲 O(n)。以下爲循序搜尋程式，使用一層迴圈求解，該迴圈執行 n 次，所以程式的複雜度爲 O(n)。

```
a = [6, 7, 4, 5, 2, 8, 3]
n = 7
max = 0
for i in range(n):
    if max < a[i]:
        max = a[i]
print(max)
```

評估程式的複雜度 (5/9)

(4) O(n*log(n))

以下為合併排序，mergesort 函式每次將資料拆成一半，合併排序的 mergesort 的遞迴深度為 O(log(n))，而 merge 函式每一層都需要 O(n)，所以合併演算法效率為 O(n*log(n))，排序單元會詳細介紹合併排序演算法。

```
a=[60, 50, 44, 82, 55, 24, 99, 33]
tmp=[0, 0, 0, 0, 0, 0, 0, 0]
def merge(L, M, R):
    left=L
    right=M+1
    i=L
    while (left <= M) and (right <= R):
        if a[left]<a[right]:
            tmp[i]=a[left]
            left = left + 1
        else:
            tmp[i]=a[right]
            right = right + 1
        i = i + 1
    while left<=M:
        tmp[i]=a[left];
        i = i + 1
        left = left + 1
    while right<=R:
        tmp[i]=a[right]
        i = i + 1
        right = right + 1
    for i in range(L, R+1):
        a[i]=tmp[i]
def mergesort(L,R):
    if L < R:
        M=(L+R)//2
        mergesort(L,M)
        mergesort(M+1,R)
        merge(L,M,R)
        print("L=", L, "M=", M," R=",R)
        for item in a:
            print(item,' ', end='')
        print()
```

評估程式的複雜度 (6/9)

(5) $O(n^2)$

以下程式爲九九乘法表，兩層迴圈各執行 n 次，所以程式的複雜度爲 $O(n^2)$。

```
n = 10
for i in range(1, n):
    for j in range(1, n):
        print(i, '*', j, '=', i*j, '\t', sep='',end='')
    print()
```

評估程式的複雜度 (7/9)

(6) $O(n^3)$

以下程式為 Floyd Warshall 找尋最短路徑演算法的部分程式碼，三層迴圈各執行 n 次，所以程式的複雜度為 $O(n^3)$。

```
n = len(City)
for k in range(n)):
    for i in range(n):
        for j in range(n):
            if dis[i][k]==1000000 or dis[k][j]==1000000:
                continue
            dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j])
```

評估程式的複雜度 (8/9)

(7) $O(2^n)$

以下為費式數列的遞迴程式，因為 F(k) 需要遞迴求解 F(k - 1) 與 F(k - 2)，若 k 值很大時，k 值下降速度很慢，相當於一分為二，程式效率為 $O(2^n)$。

```
def F(k):
    if k == 0 or k == 1:
        return  1
    else:
        return  F(k - 1) + F(k - 2)
k = int(input("請輸入 k 值？"))
result = F(k)
print("F(", k, ")=", result)
```

評估程式的複雜度 (9/9)

(8) O(n!)

以下程式列出 n 個數值的各種排列，有 n! 種排列方式，所以程式顯示至少 n! 次，此程式效率至少 O(n!)，甚至 $O(n^n)$。

```
def perm(curStep):
    if curStep == n:
        for i in range(n):
            print(num[step[i]], " ", end="")
        print()
    else:
        for i in range(n):
            success = True
            for j in range(curStep):
                if step[j] == i:
                    success = False
                    break
            if success:

                step[curStep] = i
                perm(curStep+1)


perm(0)
```

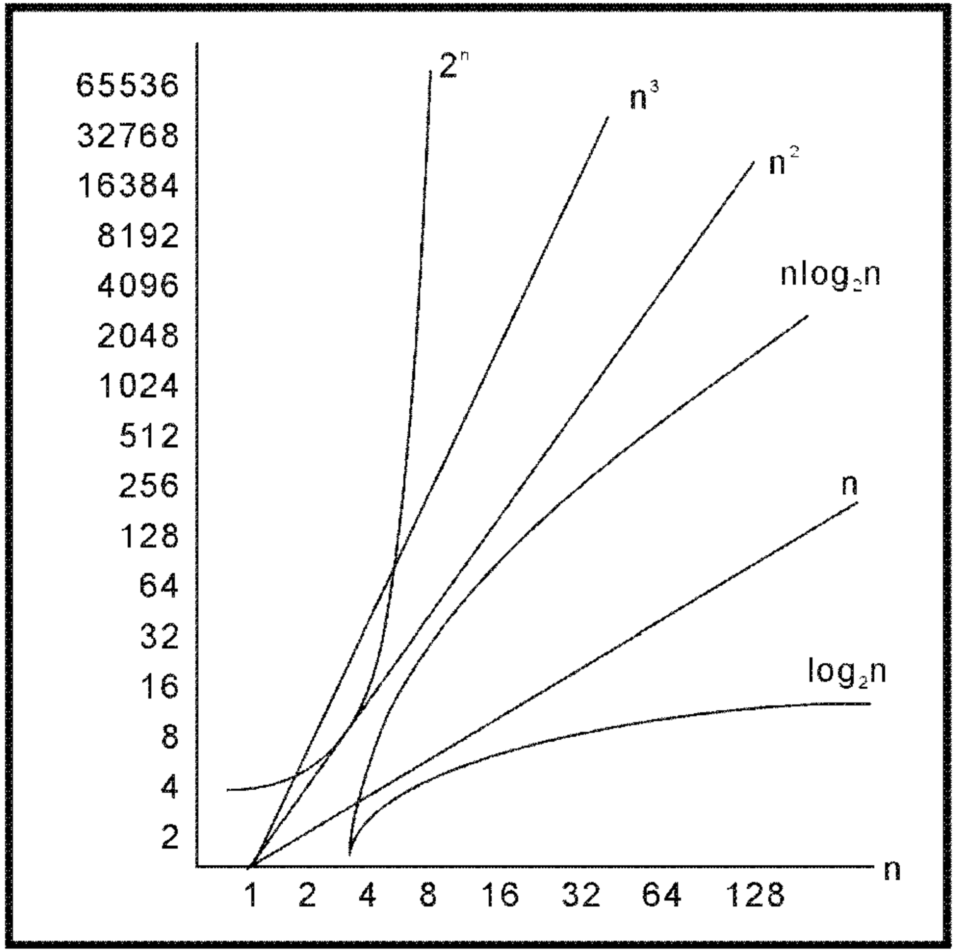
在撰寫程式時，若遇到程式效率爲 $O(2^n)$ 或 O(n!)，當 n 值不大還可以接受；當 n 值較大時，就需要考慮以更有效率的演算法撰寫程式。

Big-O (1/3)

- 一般常見的Big-O有下列幾種情形：
  - O(1) 稱為常數時間(constant)
  - O(log n) 稱為次線性時間(sub-linear)
  - O(n) 稱為線性時間(linear)
  - O(n log n) 稱為n logn n
  - O(n^2) 稱為平方時間(quadratic)
  - (n^3) 稱為立方時間(cubic)
  - O(2^n) 稱為指數時間(exponential)
  - O(n!) 稱為階層時間

Big-O (2/3)

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O(n!)$



尚可接受

需要改進

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(n^k) < O(2^n) < O(3^n) < O(n!)$

常數時間　對數時間　線性時間　次方時間　指數時間　階乘時間

Big-O (3/3)

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2147483648