Algorithms

Week 4- Time and Space Complexity
(時間&空間複雜度)

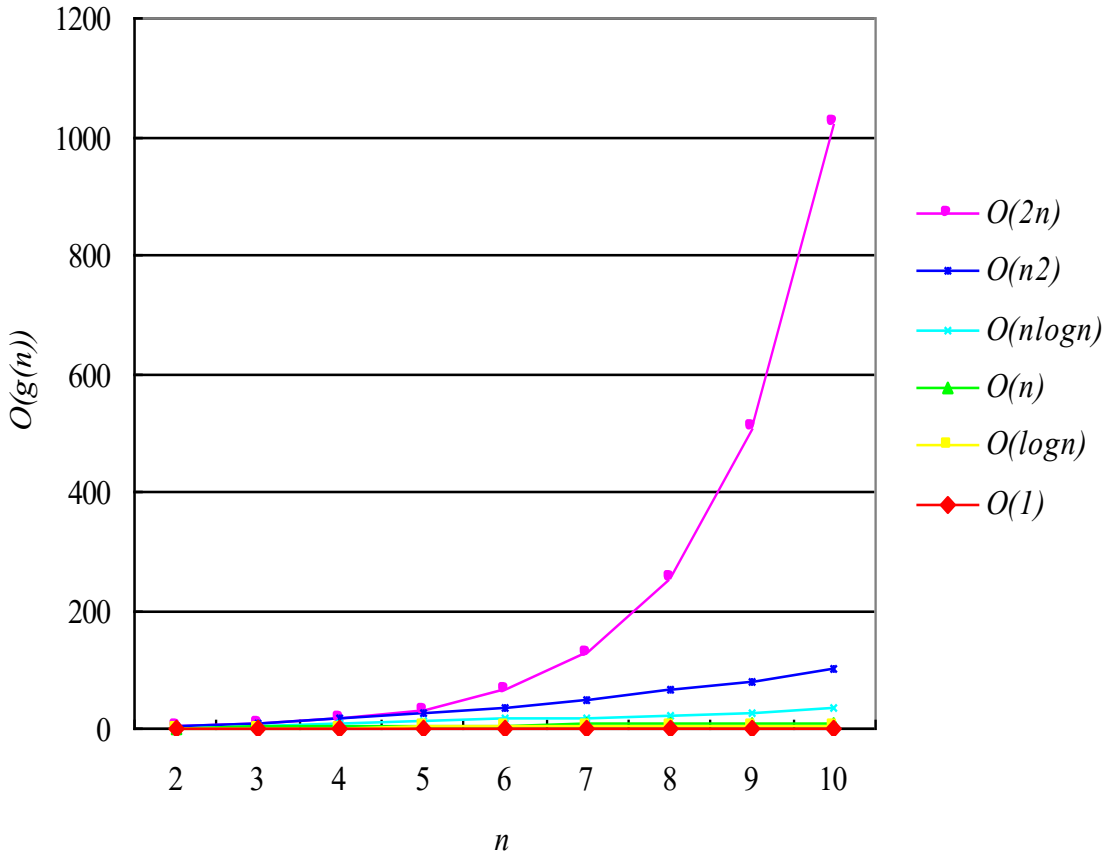
Pei-Yu Cheng

https://109nutn.github.io/algorithm/

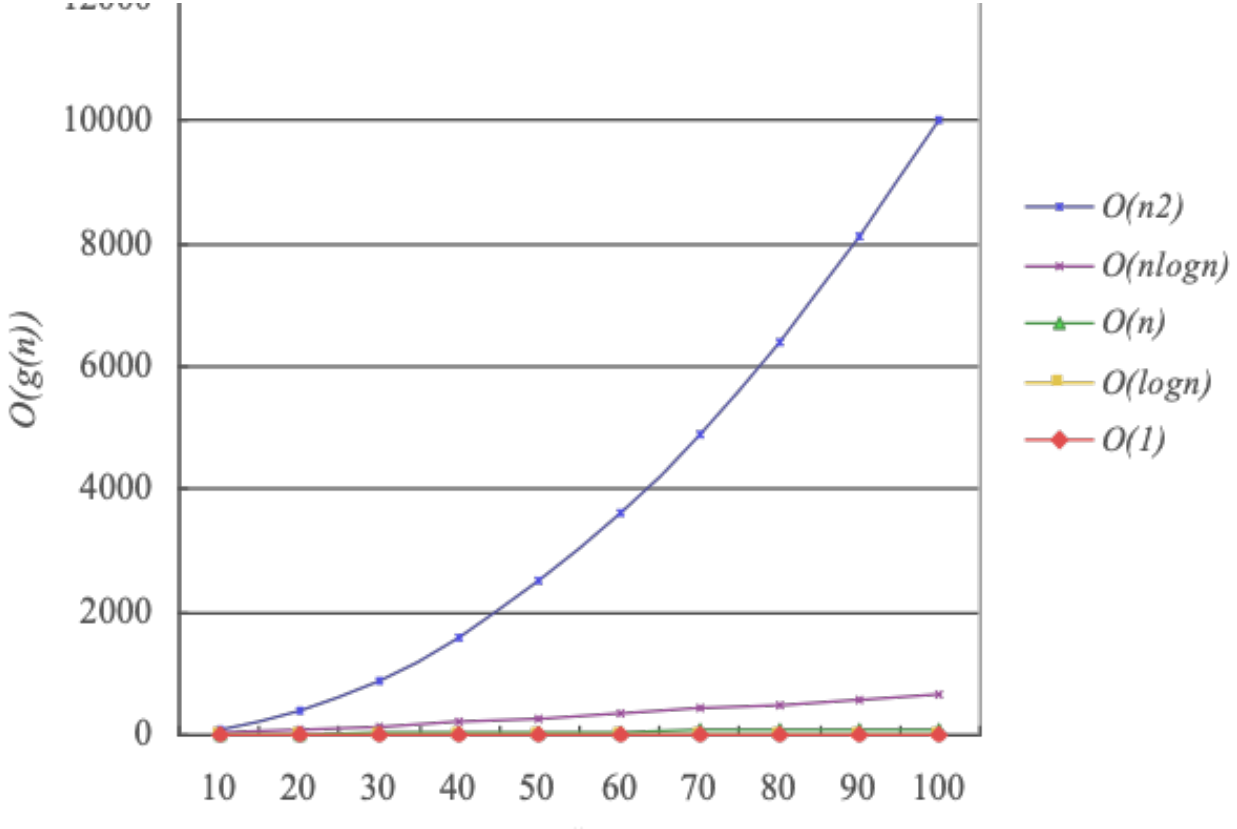
時間複雑度 → 概算

Big-O 表示法

- O(1)稱為常數時間，即不論演法的步驟須需要多少指令，只要不像迴圈般重複執行，皆視為常數時間
- O(n)稱為線性時間，取其執行步驟的增加趨勢與n的增加趨勢為線性關係之意
- $O(n^2)$為平方時間
- $O(2^n)$則稱為指數時間。


1200
1000
800
600
400
200
0
O(g(n))
2
3
4
5
6
7
8
9
10
n
O(2n)
O(n2)
O(nlogn)
O(n)
O(logn)
O(1)

n≤10 ，O函數值的變化



10 ≤ n ≤ 100，O函數值的變化

Big-O 表示法

- O(logn)的演算法比O(n)來得有效率
- O(n)比O(n^2)來得有效率。
- 如果n足夠大時 $\rightarrow 1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$

Big-O 表示法

print (“abc”) → ? O(1)

print (“abc”)
print (“abc”) → ? O(1)
print (“abc”)

Big-O 表示法

```
for (1 to N) {
    print(“abc”)   → ?
}
```

Big-O 表示法

```
for (1 to N) {
      print(“abc”)
}
for (1 to N) {
      print(“abc”)
}
for (1 to N) {
      print(“abc”)
}
```

→ ?

Big-O 表示法

```
for (1 to N) {
    for (1 to N) {
        print(“abc”)
    }
}
```

→ ? $O(n^2)$

Big-O 表示法

```
for (1 to N; step =N/2) {
    print(“abc”)
}
```

→ ? O(n)

O(n) 線性時間

```
for (i = 0; i < N; i++)  {
        k=k+1;
}
```

→? O(1)

→? O(n)

for loop 執行N次，全部執行時間是 ? O(n)

O(n²) 二次方

完成程式碼需要N的平方次

```
for (i = 0; I < N; i++)   {          → ?
     for (j = 0; j < M; j++)   {     → ?
          sequence of statements     →?
               }
          }
```

外面loop 執行N次，裡面執行M次

全部執行時間是 → ?

O(n²) 二次方

```
for (i = 0; I < N; i++)   {          → ?
    for (j = 0; j < N; j++)   {      → ?
        sequence of statements       →?
            }
        }
```

O(n) N

O(n²) N

O(1)

全部執行時間是 → ?

O(n²)

$O(n^2)$ 二次方

```
for (i = 0; I < N; i++)   {          → ?
    for (j = i+1; j < N; j++)   {    → ?
        sequence of statements       →?
            }
        }
```

全部執行時間是 → ?

O(1)

$\frac{n \times (n-1)}{2}$

$O(n^2)$

用 Big-O 來表示最多次(概略)的執行次數

3n+1= ? O(n)

3n+3= ? O(n)

50n+5= ? O(n)

$10n^2+5n+1$ = ? $O(n^2)$

$5*2^n+n^2$ = ? $O(2^n)$

Big-O 運算

$O(4) + O(N) = ?$

$O(\log N) + O(\log N) + O(\log N) = ?$

$O(N) + O(N^2) = ?$

O(N)

O(logN)

O(N²)

空間複雜度 (Space Complexity)

- 定義：演算法所需要消耗的儲存記憶體資源
  → 電腦執行演算法所需要耗費的空間成本(記憶體)

- 計算和表示方法與時間複雜度類似
- 一般以複雜度的漸近性來表示(asymptotic analysis)
- S(n)

- 空間複雜度比較常用的有：O(1)、O(n)、$O(n^2)$

空間複雜度 (Space Complexity)

```
int i = 1;
int j = 2;
++i;
j++;
int m = i + j;
```

i、j、m 所分配的空間不管程式跑了幾次，都不會影響使用的變數數量，故該函式的空間複雜度為 S(n) = O(1)

空間複雜度 (Space Complexity)

```
int [ ] m = new int [n]
for(i=1; i<=n; ++i)
{
   j = i;
   j++;
}
```

第一行new了一個 [n] 出來，這個數據佔用的大小為n

2-6行，雖然有循環，但沒有再分配新的空間

因此，這段代碼的空間複雜度主要看第一行，即 S(n) = O(n)

時間與空間複雜度 （Time and Space Complexity）

「時間複雜度」與「空間複雜度」可以相互權衡(trade off)

- 讓程式多用一些記憶體空間來多記一些資訊，就可以省去一些重複的運算來加速程式的執行時間（空間換時間）

- 在沒有多餘可用的記憶體資源下，可透過把一些原本靠記憶體存儲的資訊改用重複計算的方式來取得 （時間換空間）

時間與空間複雜度 （Time and Space Complexity）

例如：如何判斷某年是不是閏年？

- 方法一

  寫一個演算法，每給一個年份，就可以通過該演算法計算得到是否閏年的結果

- 方法二

  先建立一個所有年份的陣列，然後把所有的年份按下標的數字對應；
  如果是閏年，則此陣列元素的值是1，如果不是元素的值則為0。
  這樣，所謂的判斷某一年是否為閏年就變成了查詢這個陣列某一個元素的值的問題

時間與空間複雜度 （Time and Space Complexity）

例如：如何判斷某年是不是閏年？

- 方法一：

  相比起第二種來說很明顯非常節省空間，但每一次查詢都需要經過一系列的計算才能知道是否為閏年。

- 方法二：

  第二種方法雖然需要在記憶體裡儲存所有年份的陣列，但是每次查詢只需要一次索引判斷即可。

空間換時間 vs 時間換空間 到底哪一種方法好？

Assessments-3

- 設想一個與上面例子相似的情境
- 寫出兩種算法 (空間換時間 vs 時間換空間)
- 檔名 取為 ts_analysis，上傳至 GitHub

Q&A

Pei-Yu Cheng

peiyu.cheng.tw@gmail.com