Complexity

上課補充 by ZCKevin
Credit by Howard41436, double

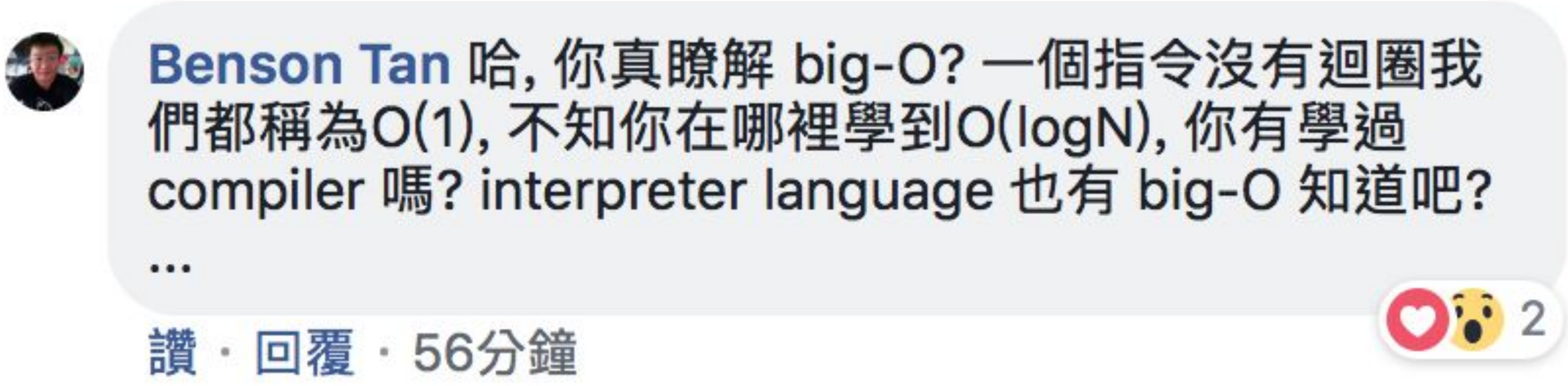
- 影片看了嗎
- Q&A

今天要學的東西

- 今天的課以理論為主，不太會寫到題目
- 但是對於複雜度的知識絕對是寫程式最重要的一環
- 大家都怎麼算複雜度？

今天要學的東西

- 今天的課以理論為主，不太會寫到題目
- 但是對於複雜度的知識絕對是寫程式最重要的一環
- 大家都怎麼算複雜度？


Benson Tan 哈, 你真瞭解 big-O? 一個指令沒有迴圈我們都稱為O(1), 不知你在哪裡學到O(logN), 你有學過 compiler 嗎? interpreter language 也有 big-O 知道吧?
...

2

讚 · 回覆 · 56分鐘

今天要學的東西

- 今天的課以理論為主，不太會寫到題目
- 但是對於複雜度的知識絕對是寫程式最重要的一環
- 大家都怎麼算複雜度？
- “沒有迴圈就是O(1)，一層迴圈就是O(n)，兩層就O(n^2)”
- By Foxconn Specialist

今天要學的東西

- 今天的課以理論為主，不太會寫到題目
- 但是對於複雜度的知識絕對是寫程式最重要的一環
- 大家都怎麼算複雜度？
- “沒有迴圈就是O(1)，一層迴圈就是O(n)，兩層就O(n^2)”
- By Foxconn Specialist
- 真的是如此嗎？

複雜度的正確算法

- 數迴圈有幾層的優缺點:
- 優點:大部分的程式我們大概都可以這樣算啦
- 缺點:
- 1.有些操作可能不是O(1)
- 2.無法計算遞迴的時間複雜度
- 3.可能會錯估均攤複雜度

複雜度的正確算法

- 首先，先來討論複雜度的正確算法
- 正式的數學定義手寫作業有，也會讓大家證一些東西
- 不過在99%的時候，複雜度都這樣算就好了：
- 計算總共需要的操作數，留下量級最大那一項，常數去掉
- ex: $O(3n^2\log n + 2n^2 + 4n + \log n) = O(n^2\log n)$

案例1

- 計算a的n次方模10^9+7:
- Foxconn specialist:
- return a**n % 1000000007
- 沒有迴圈，所以O(1)
- 實測時間

案例1

```python
from time import *
Time_1 = time()
a, n, mod = 3, 23, 1000000007
res_1 = a ** n % mod
Time_2 = time()
n = 1000000
res_2 = a ** n % mod
Time_3 = time()
print("Step 1 takes ", Time_2 - Time_1)
print("Step 2 takes ", Time_3 - Time_2)
```

案例1

```python
from time import *
Time_1 = time()
a, n, mod = 3, 23, 1000000007
res_1 = a ** n % mod
Time_2 = time()
n = 1000000
res_2 = a ** n % mod
Time_3 = time()
print("Step 1 takes ", Time_2 - Time_1)
print("Step 2 takes ", Time_3 - Time_2)
```

```
Step 1 takes  6.67572021484375e-06
Step 2 takes  0.06563568115234375
```

案例1

- 事實上，計算a的n次方時，通常是在n很小，或著是在做a或n為浮點數的運算時，我們才會用內建的 ** 運算，且將複雜度當作O(1)
- 當計算整數a的n次方時，內建的 ** 運算似乎變成與n成線性時間，複雜度應為O(n)
- 未來會學到如何將複雜度降為O(logn)（快速冪）

案例2

- 估算遞迴的複雜度
- gcd(p, q):

```
if q = 0:
    return p
else:
    return gcd(q, p % q)
```

- 這種自己呼叫自己的函式，沒有迴圈可以數，複雜度要怎麼算？

案例2

- 估算遞迴的複雜度
- fib(n):

```
    if n = 1 or 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

- 這種自己呼叫自己的函式，沒有迴圈可以數，複雜度要怎麼算？

案例3

- 模擬一個stack，有3種操作
- 1. push(x)
- 2. top()
- 3. pop(k) ： 連續pop掉k個物品，其中k<=stack的大小
- 執行n次操作的總複雜度是多少？
- 每個操作的時間複雜度是多少？

案例3

- 模擬一個stack，有3種操作
- 1. push(x)
- 2. top()
- 3. pop(k) ： 連續pop掉k個物品，其中k<=stack的大小
- 執行n次操作的總複雜度是多少？$O(n*n)=O(n^2)$?
- 每個操作的時間複雜度是多少？O(1),O(1),O(n)

案例3

- 模擬一個stack，有3種操作
- 1. push(x)
- 2. top()
- 3. pop(k) ： 連續pop掉k個物品，其中k<=stack的大小
- 每個操作的時間複雜度是多少？O(1),O(1),O(n)
- 執行n次操作的總複雜度是多少？$O(n*n)=O(n^2)$?
- pop掉的物品數<=push進的物品數<=n
- 均攤複雜度O(n)

案例4

- 底下的程式會計算對於所有 1 <= i <= n, i 有幾個因數

```
vector<int> number_of_factor(int n) {
  vector<int> cnt(n + 1);
  for (int i = 1;i <= n;++i) {
    for (int j = i;j <= n;j += i)
      ++cnt[j];
  }
  return cnt;
}
```

案例4

- 底下的程式會計算對於所有 1 <= i <= n, i 有幾個因數
- 複雜度 O(N log N)

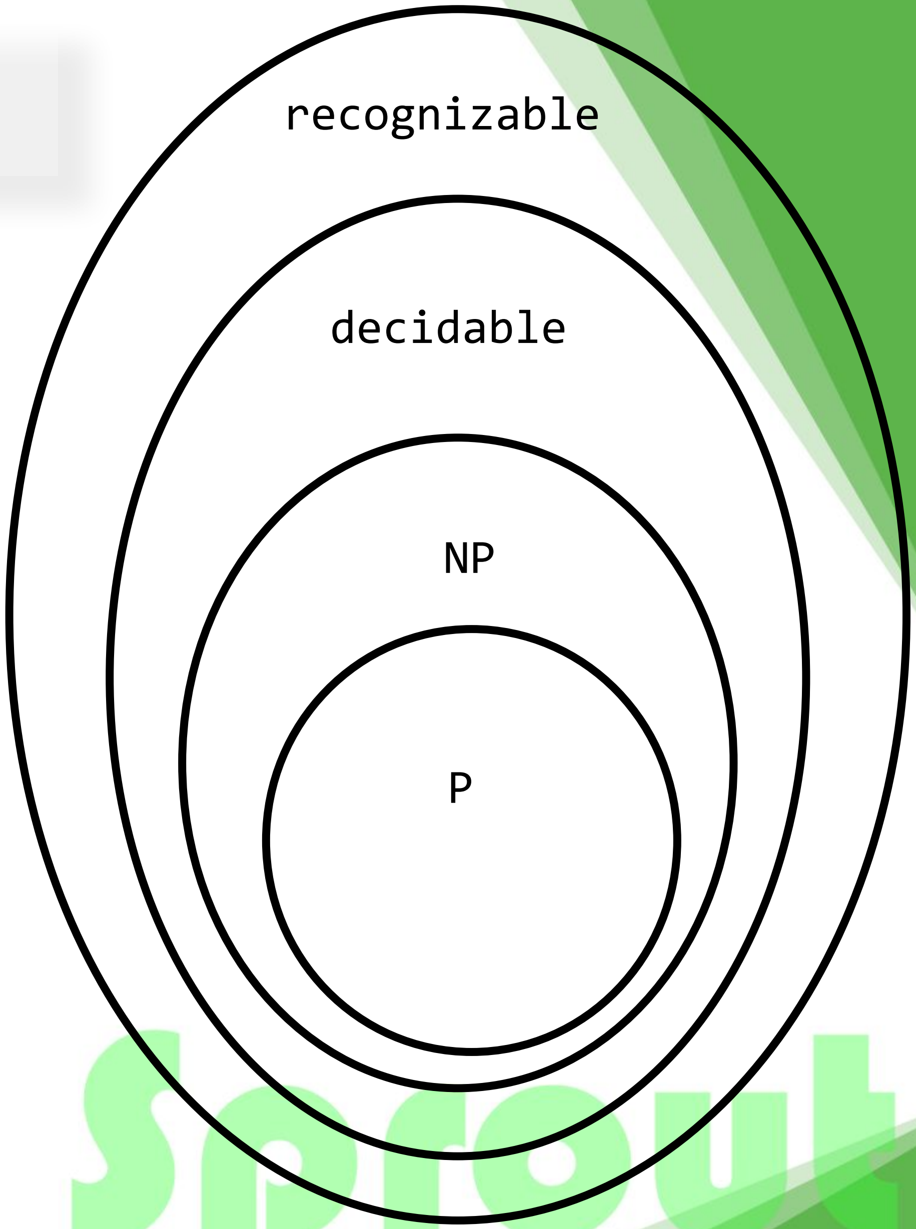
```
vector<int> number_of_factor(int n) {
  vector<int> cnt(n + 1);
  for (int i = 1;i <= n;++i) {
    for (int j = i;j <= n;j += i)
      ++cnt[j];
  }
  return cnt;
}
```

複雜度的重要性

- 在競賽程式中，複雜度計算通常是為了讓我們判斷這個演算法寫了會不會TLE
- 我們可以假設機器每秒大概跑 5e8 左右的操作
- 這件事也告訴我們，除非複雜度真的在 5e8 左右，否則演算法的常數不是很重要，如果複雜度相同挑最好寫的方式寫就好

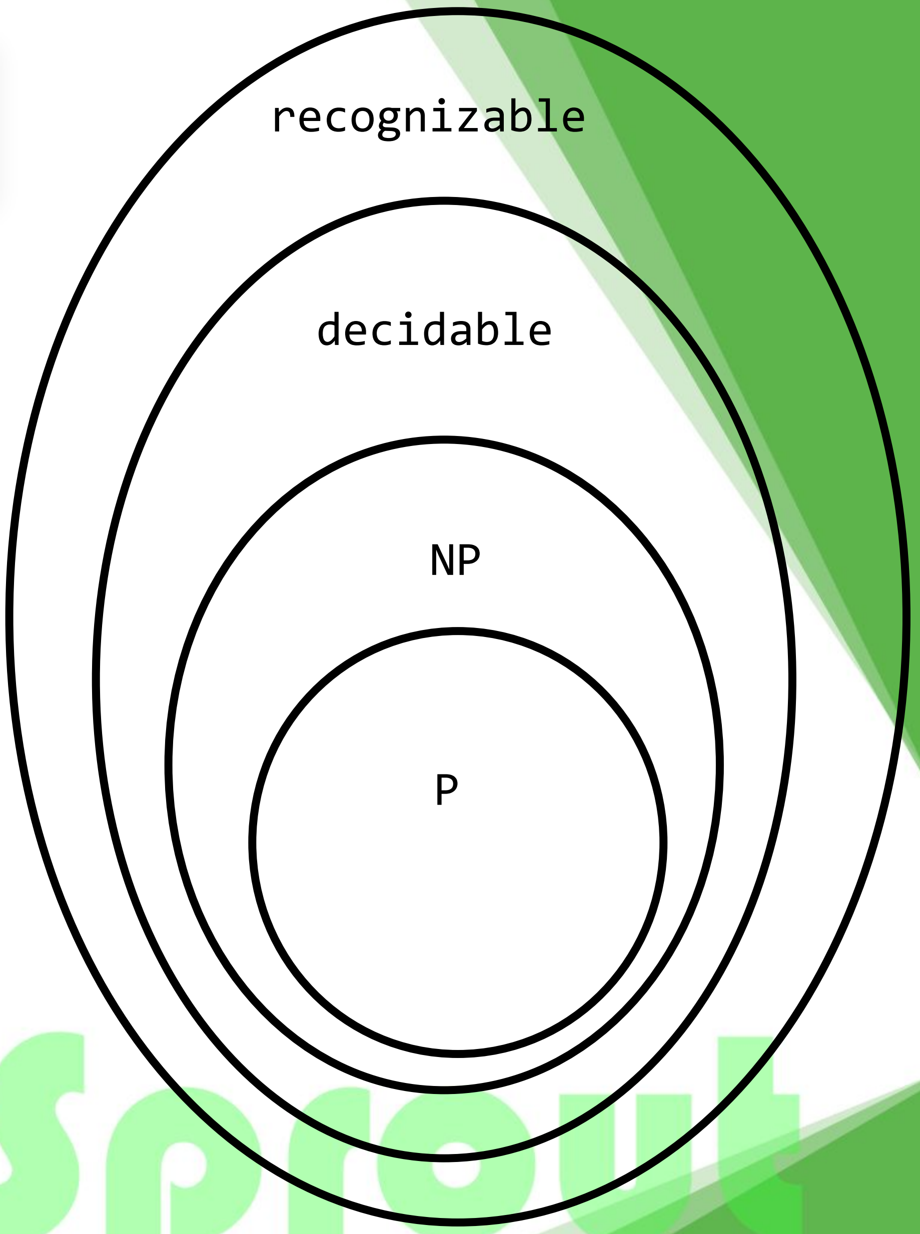
計算理論中的複雜度類

- recognizable
- decidable
- nondeterministic polynomial
- polynomial



計算理論中的複雜度類

- recognizable
  對於答案是yes的輸入，能夠輸出yes
  對於答案是no的輸入，則不一定跑的完
- decidable
- nondeterministic polynomial
- polynomial

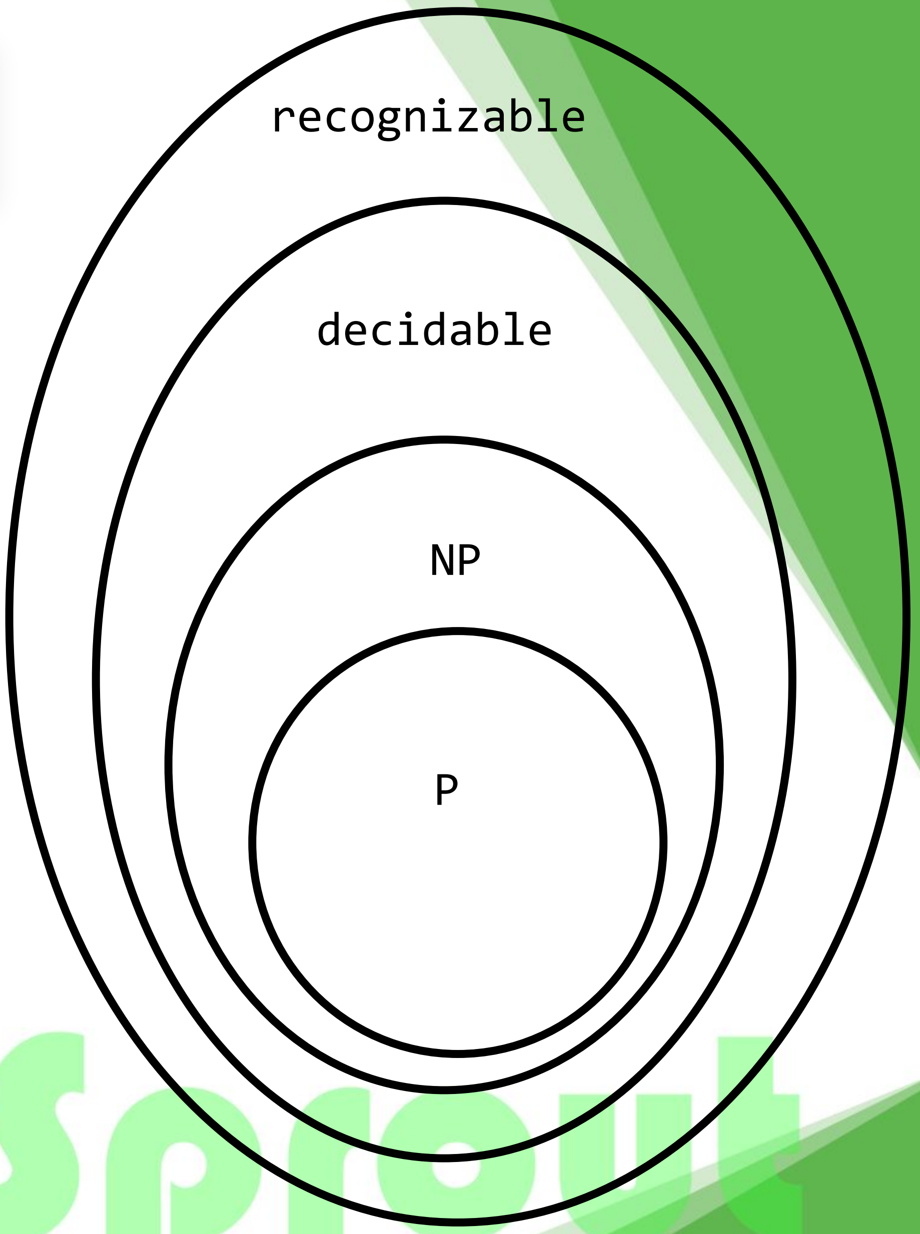


計算理論中的複雜度類

- recognizable
- decidable

  對於答案是yes的，能輸出yes
  對於答案是no的，能輸出no

  此複雜度類的問題才有「演算法」
- nondeterministic polynomial
- polynomial



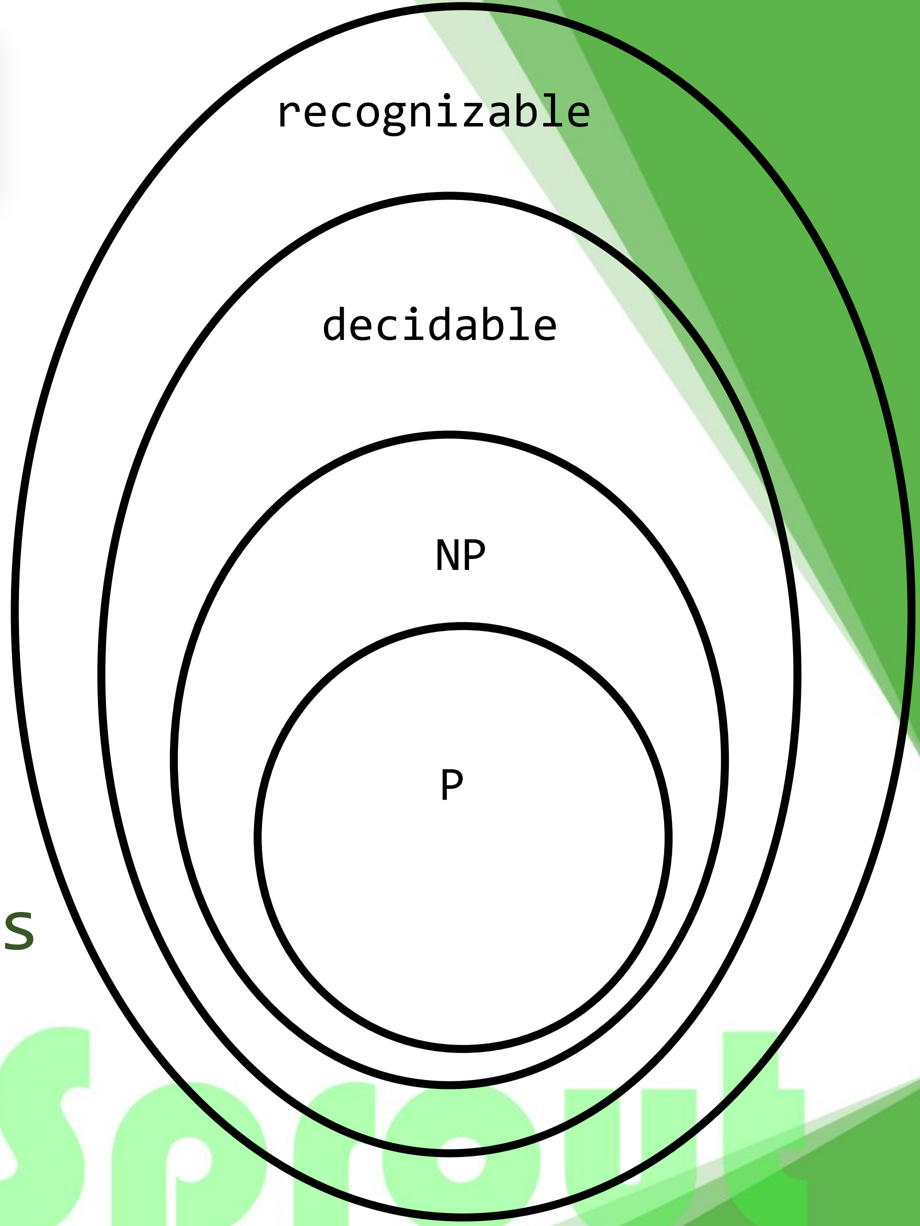
計算理論中的複雜度類

- recognizable
- decidable
- nondeterministic polynomial

  常常聽到的NP問題，並不是指非多項式時間(decidable裡面很多非多項式但也不是NP的問題)

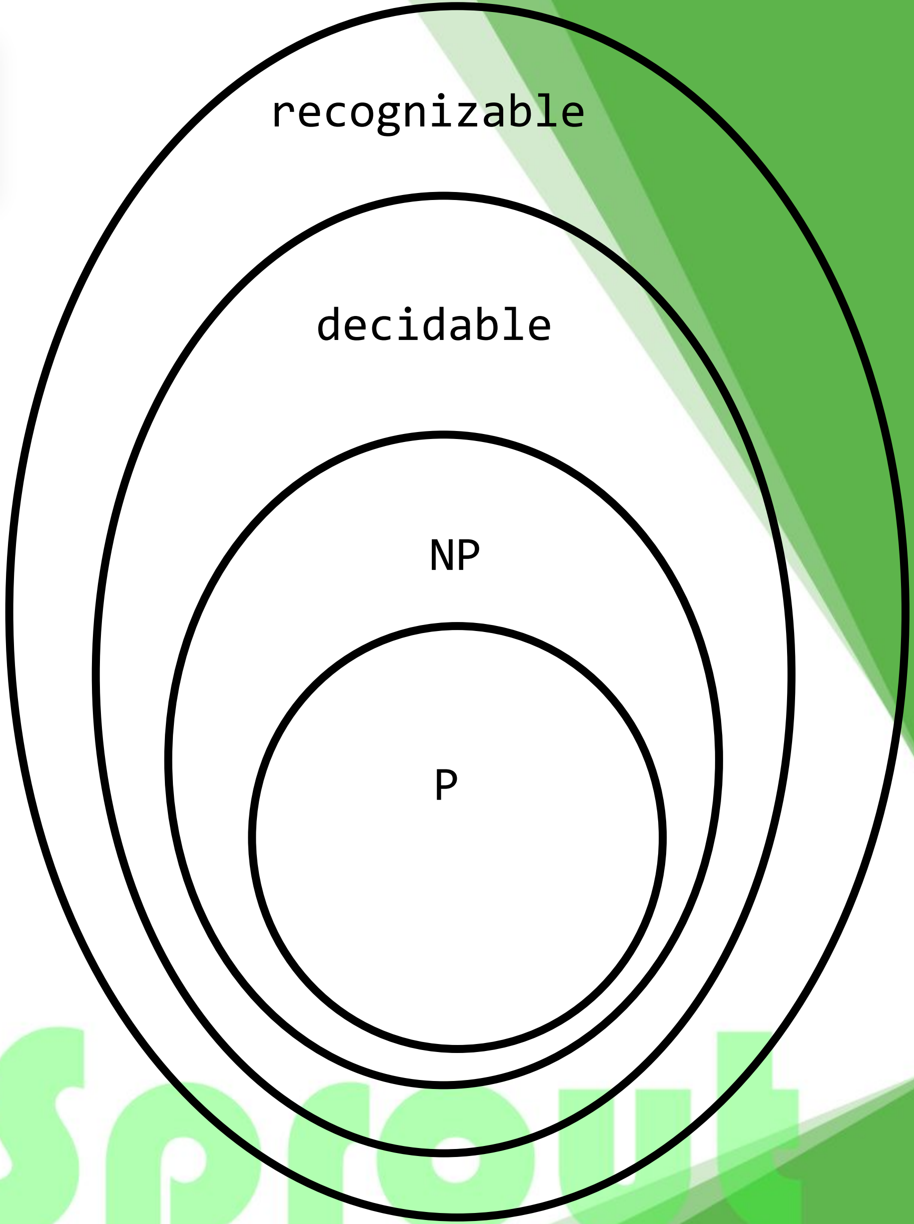
  是指如果我們多給程式一個提示，

  就有辦法在多項式時間內得知答案是yes
- polynomial



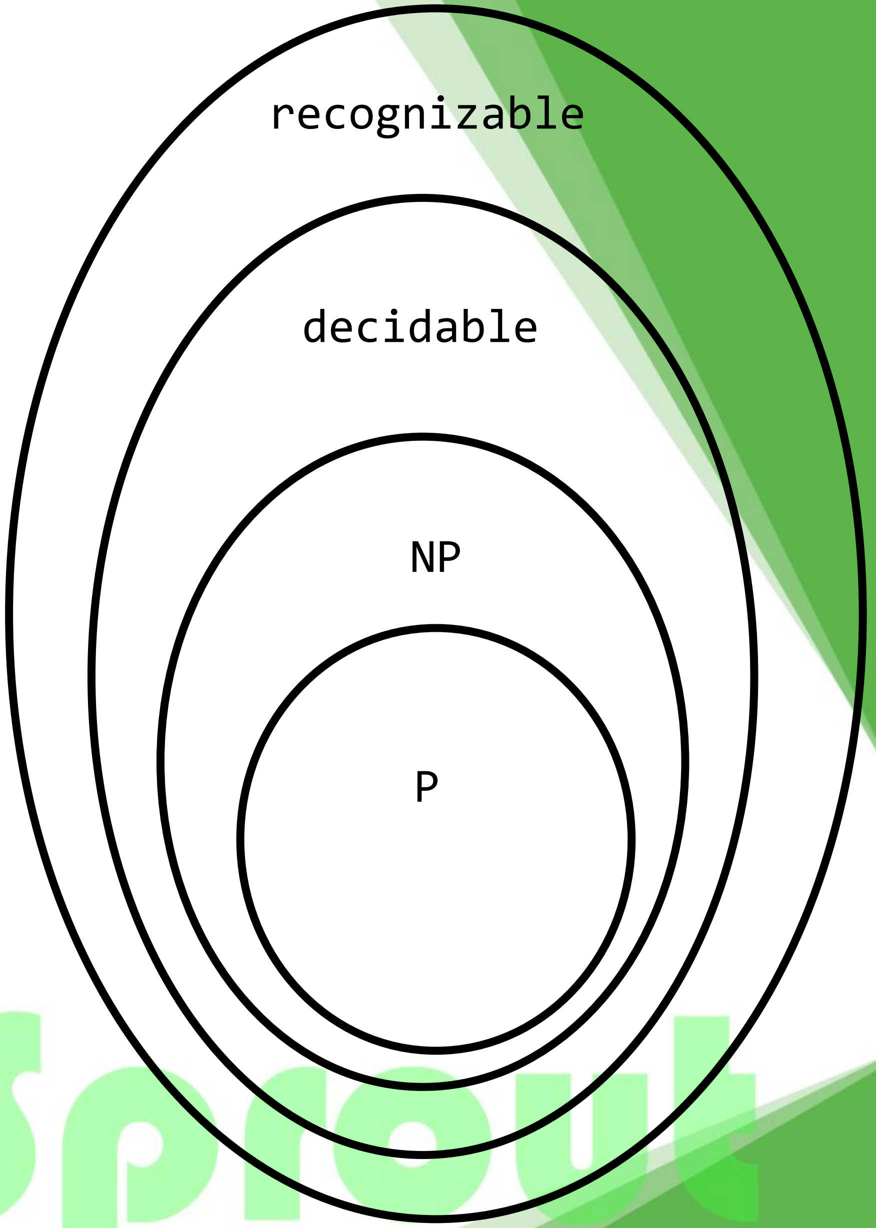
計算理論中的複雜度類

- recognizable
- decidable
- nondeterministic polynomial
- polynomial
  這個就很直觀了，是指能夠在多項式時間內判斷答案是yes還是no
  一般我們能有效率解決的問題都是P問題



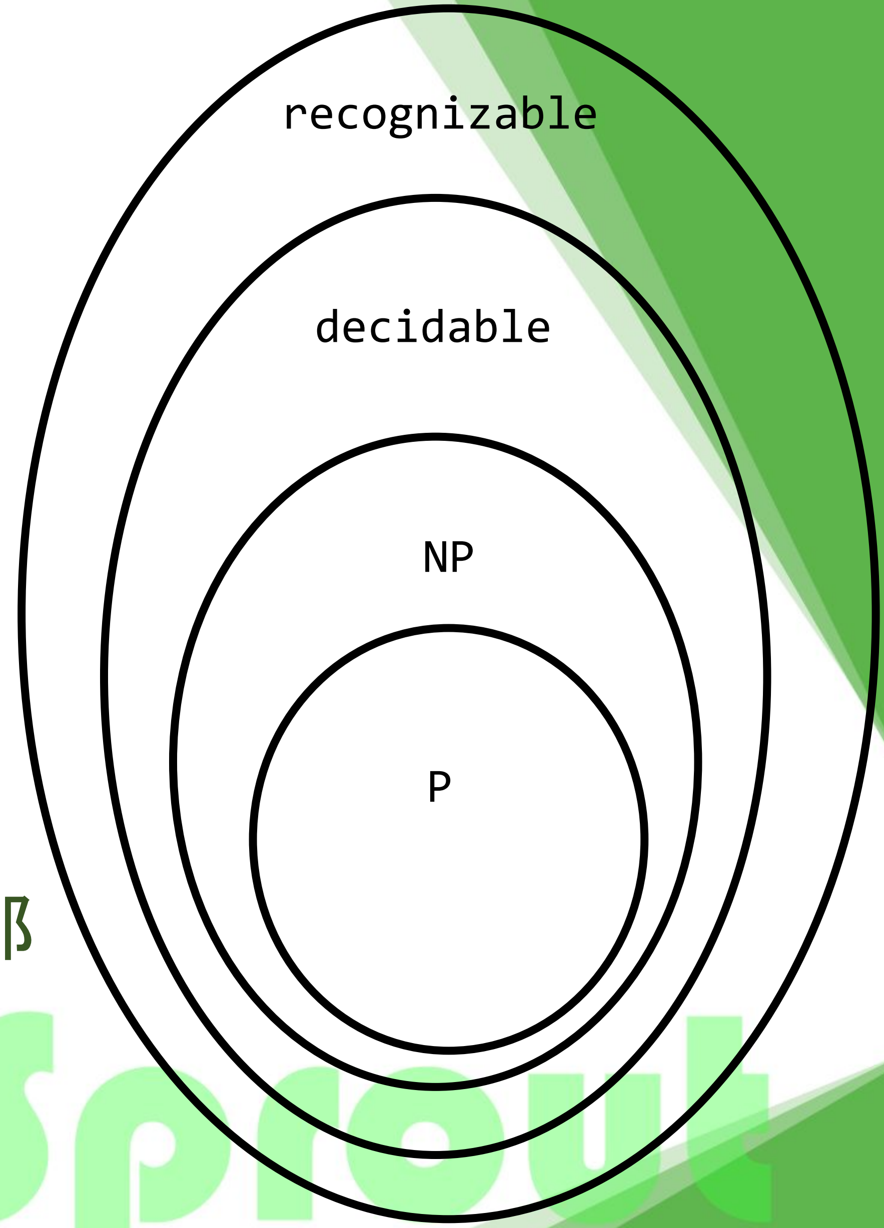
P與NP

- 所有P問題都是NP問題
- 我們可能會很想幫NP和P之間的區域取一個名字，這樣當我們發現一個問題很難，我可就可以試圖去證明它屬於那個區域



P與NP

- 所有P問題都是NP問題
- 我們可能會很想幫NP和P之間的區域取一個名字，這樣當我們發現一個問題很難，我可就可以試圖去證明它屬於那個區域
- 但目前我們不確定那個區域是否存在
- 也就是有可能P=NP
- 於是我們定義NP問題裡最難的那些問題為NP-Complete，如果有一天確定了P!=NP，那那些NPC的問題肯定不屬於P



多項式時間複雜度

- 剛剛P和NP都有提到的多項式時間，大家都知道多項式要有一個變數，那麼題目裡的變數有時候是指個數，有時候是數字範圍，有時候甚至超過一個變數，多項式時間是指哪個變數的多項式呢
- 印出1+2+...+n：
- 1.使用公式解
- 2.使用暴力法
- 上述哪些是多項式時間演算法，哪些不是呢？

多項式時間複雜度

- 剛剛P和NP都有提到的多項式時間，大家都知道多項式要有一個變數，那麼題目裡的變數有時候是指個數，有時候是數字範圍，有時候甚至超過一個變數，多項式時間是指哪個變數的多項式呢
- A： 輸入總長度(字元數)，N，的多項式
- 因此當你的複雜度裡面有代表數字大小而非個數的變數時，其實不能算是多項式演算法，因為長度為N的輸入可以表示出約10^N大小的數字。
- 這種看似是多項式時間的演算法，我們會稱其為「偽多項式時間」演算法。在題目有限制數字範圍不會太大時依然很有效率。

問題時間

- 為什麼我們剛剛都只討論yes/no問題？這樣適用於其他問題嗎？

P問題的重要性

- 當我們在歸類一個問題為P問題時，等於不在乎他的複雜度是$O(N^2)$還是$O(N^3)$之類的，只要是多項式時間就好。
- 這是因為多項式次數的差別可以透過運算速度隨著科技進步的提升，或是更多更高級的運算資源來突破。但是非多項式時間的問題我們基本上就是永遠沒辦法有效率的解決。

P問題的重要性

- 當我們在歸類一個問題為P問題時，等於不在乎他的複雜度是$O(N^2)$還是$O(N^3)$之類的，只要是多項式時間就好。
- 這是因為多項式次數的差別可以透過運算速度隨著科技進步的提升，或是更多更高級的運算資源來突破。但是非多項式時間的問題我們基本上就是永遠沒辦法有效率的解決。
- 存在無法多項式時間解決的問題並不是壞事，這可以製造computational gap
- 可以用在密碼學中
- 如果P=NP會發生什麼事？

P=?NP

- If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett.