Complexity

Lecture by LittleCube

部份內容參考自 2024 Complexity PixelCat

Sprout

課程影片

Q & A?

Sprout

漸進複雜度

Sprout

漸進複雜度

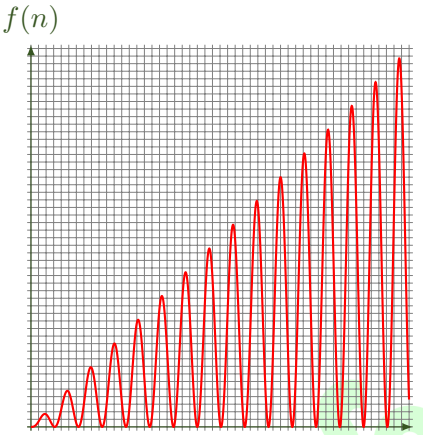
- 課程影片說複雜度是用極限去算的

漸進複雜度

- 課程影片說複雜度是用極限去算的
- 極限給大家一個直覺的定義：想要兩個函數的比例是一個常數！

Sprout

漸進複雜度



漸進複雜度

- 不過極限有一些小問題（而且太難了）

漸進複雜度

- 不過極限有一些小問題（而且太難了）
- 數學上的正式定義：

Definition Big-O Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) \in O(g(n))$ 或 $f(n) = O(g(n))$ 如果

$$\exists n_0 > 0, c > 0, \text{使得} \forall n \geq n_0, f(n) \leq c \cdot g(n)。$$

你們這次的手寫作業會有數學上的完整定義！

Sprout

漸進複雜度

- 不過極限有一些小問題（而且太難了）
- 數學上的正式定義：

Definition Big-O Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) \in O(g(n))$ 或 $f(n) = O(g(n))$ 如果

$$\exists n_0 > 0, c > 0, \text{使得} \forall n \geq n_0, f(n) \leq c \cdot g(n)。$$

你們這次的手寫作業會有數學上的完整定義！

- 直覺概念：n 夠大的時候 $cg(n)$ 都可以把 $f(n)$ 壓下去！

漸進複雜度

當然漸進複雜度家族不是只有 Big-O 這個人！

符號	名稱	直覺概念
$f(n) = o(g(n))$	Little O	$<$
$f(n) = O(g(n))$	Big O	$\leq$
$f(n) = \Theta(g(n))$	Big Theta	$\approx$
$f(n) = \Omega(g(n))$	Big Omega	$\geq$
$f(n) = \omega(g(n))$	Small Omega	$>$

漸進複雜度

Definition Big-O Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) = O(g(n))$ 如果

$$\exists n_0 > 0, c > 0, \text{使得} \forall n \geq n_0, f(n) \leq c \cdot g(n)。$$

Definition Big-Omega Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) = \Omega(g(n))$ 如果

$$\exists n_0 > 0, c > 0, \text{使得} \forall n \geq n_0, f(n) \geq c \cdot g(n)。$$

漸進複雜度

Definition Big-Theta Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) = \Theta(g(n))$ 如果

$$\exists n_0 > 0, c_1 > 0, c_2 > 0, \text{使得} \forall n \geq n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)。$$

Corollary

$$f(n) = O(g(n)), f(n) = \Omega(g(n)) \iff f(n) = \Theta(g(n))$$

漸進複雜度

Definition Little-O Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) = o(g(n))$ 如果

$$\forall c > 0, \exists n_0 > 0, \text{使得} \forall n \geq n_0, f(n) \leq c \cdot g(n)。$$

Definition Little-Omega Notation

對於只有在正整數上有定義的函數 f, g，我們說 $f(n) = \omega(g(n))$ 如果

$$\forall c > 0, \exists n_0 > 0, \text{使得} \forall n \geq n_0, f(n) \geq c \cdot g(n)。$$

無論常數怎麼選，$f(n)$ 在 n 夠大都一定會打輸或打贏

漸進複雜度

n vs n^2 ?

n^2 vs $n \log n$?

2^n vs n^2 ?

2^n vs 3^n ?

n^n vs $n!$?

$2n$ vs n ?

Sprout

漸進複雜度

$$
\begin{aligned}
n &= o(n^2) \\
n^2 &= \omega(n \log n) \\
2^n &= \omega(n^2) \\
2^n &= o(3^n) \\
n^n &= \omega(n!) \\
2n &= \Theta(n)
\end{aligned}
$$

Sprout

漸進複雜度

誰比較大？

n vs 2^n ?

漸進複雜度

誰比較大？

n vs 2^n ?

n^{10} vs 2^n ?

漸進複雜度

誰比較大？

$$
\begin{array}{lll}
n & \text{vs} & 2^n \quad ? \\
n^{10} & \text{vs} & 2^n \quad ? \\
n^{10} & \text{vs} & 1.1^n \quad ?
\end{array}
$$

漸進複雜度

誰比較大？

$$
\begin{array}{ccc}
n & \text{vs} & 2^n \\
n^{10} & \text{vs} & 2^n \\
n^{10} & \text{vs} & 1.1^n \\
n^{1000000000} & \text{vs} & 1.0000000001^n
\end{array}
\quad ? \quad ? \quad ? \quad ?
$$

Sprout

漸進複雜度

誰比較大？

$$
\begin{array}{ccc}
n & \text{vs} & 2^n \quad ? \\
n^{10} & \text{vs} & 2^n \quad ? \\
n^{10} & \text{vs} & 1.1^n \quad ? \\
n^{1000000000} & \text{vs} & 1.0000000001^n \quad ?
\end{array}
$$

多項式時間的演算法跟**指數**時間的演算法相比，複雜度總是比較好

漸進複雜度

誰比較大？

$$n \quad \text{vs} \quad \log n \quad ?$$

漸進複雜度

誰比較大？

$$n \quad \text{vs} \quad \log n \quad ?$$
$$n^{0.1} \quad \text{vs} \quad \log n \quad ?$$

漸進複雜度

誰比較大？

$$
\begin{array}{ccc}
n & \text{vs} & \log n \\
n^{0.1} & \text{vs} & \log n \\
n^{0.1} & \text{vs} & \log^{10} n
\end{array}
\quad ? \quad ? \quad ?
$$

Sprout

漸進複雜度

誰比較大？

$$
\begin{array}{ccc}
n & \text{vs} & \log n \\
n^{0.1} & \text{vs} & \log n \\
n^{0.1} & \text{vs} & \log^{10} n \\
n^{0.0000000001} & \text{vs} & \log^{1000000000} n
\end{array}
\quad ? \quad ? \quad ? \quad ?
$$

Sprout

漸進複雜度

誰比較大？

$$
\begin{array}{ccc}
n & \text{vs} & \log n \\
n^{0.1} & \text{vs} & \log n \\
n^{0.1} & \text{vs} & \log^{10} n \\
n^{0.0000000001} & \text{vs} & \log^{1000000000} n
\end{array}
\quad ? \quad ? \quad ? \quad ?
$$

多項式時間的演算法跟**對數**時間的演算法相比，複雜度總是比較差

分析複雜度

Sprout

Word RAM Model

- 最接近現代電腦的模型

Sprout

Word RAM Model

- 最接近現代電腦的模型
- 1 單位時間可以做各種運算以及存取一個指標

Sprout

Word RAM Model

- 最接近現代電腦的模型
- 1 單位時間可以做各種運算以及存取一個指標
- 假如輸入的大小是 n，一個 word 至少有 $w = \Omega(\log n)$ 個 bit（通常會假設 $\Theta(\log n)$）

算算複雜度

樸素方陣乘法

```
using namespace std;
typedef vector<vector<int>> matrix;
matrix mult(int n, matrix a, matrix b /* n×n */)
{
    matrix c = vector(n, vector(n, 0));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
    return c;
}
```

算算複雜度

樸素方陣乘法

```
using namespace std;
typedef vector<vector<int>> matrix;
matrix mult(int n, matrix a, matrix b /* n×n */)
{
    matrix c = vector(n, vector(n, 0));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
    return c;
}
```

$O(n^2)$

$O(n)$

$O(n)$

$O(n)$

$O(1)$

算算複雜度

樸素方陣乘法

```
using namespace std;
typedef vector<vector<int>> matrix;
matrix mult(int n, matrix a, matrix b /* n×n */)
{
    matrix c = vector(n, vector(n, 0));
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
    return c;
}
```

$O(n^2)$

$O(n)$

$O(n)$

$O(n)$

$O(1)$

總時間複雜度 $O(n^2) + O(n) \times O(n) \times O(n) = O(n^3)$

算算複雜度

斜的 for 迴圈

```
using namespace std;
long long calc(int n, vector<vector<int>> a /* n×n */)
{
    long long sum = 0;
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            sum += a[i][j];
    return sum;
}
```

算算複雜度

斜的 for 迴圈

```
using namespace std;
long long calc(int n, vector<vector<int>> a /* n×n */)
{
    long long sum = 0;
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            sum += a[i][j];
    return sum;
}
```

$O(n)$

$O(n-i)$

$O(1)$

算算複雜度

斜的 for 迴圈

```
using namespace std;
long long calc(int n, vector<vector<int>> a /* n×n */)
{
    long long sum = 0;
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            sum += a[i][j];
    return sum;
}
```

$O(n)$

$O(n-i)$

$O(1)$

總時間複雜度

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$$

算算複雜度

```cpp
using namespace std;
long long binary_search(const vector<int> &a /* 非嚴格遞增 */,
                        int target)
{
    int n = (int)a.size();
    int l = 0, r = n;
    while (l < r)
    {
        int mid = (l + r) / 2;
        if (a[mid] < target) l = mid + 1;
        else                 r = mid;
    }
    return l;
}
```

算算複雜度

```
using namespace std;
long long binary_search(const vector<int> &a /* 非嚴格遞增 */,
                        int target)
{
    int n = (int)a.size();
    int l = 0, r = n;
    while (l < r)
    {
        int mid = (l + r) / 2;
        if (a[mid] < target) l = mid + 1;
        else                 r = mid;
    }
    return l;
}
```

這個不知道會跑多少次

算算複雜度

```
using namespace std;
long long binary_search(const vector<int> &a /* 非嚴格遞增 */,
                        int target)
{
    int n = (int)a.size();
    int l = 0, r = n;
    while (l < r)
    {
        int mid = (l + r) / 2;
        if (a[mid] < target) l = mid + 1;
        else                 r = mid;
    }
    return l;
}
```

這個不知道會跑多少次

這兩個狀況都會把陣列從中間切開，取某一邊

算算複雜度

```cpp
using namespace std;
long long binary_search(const vector<int> &a /* 非嚴格遞增 */,
                        int target)
{
    int n = (int)a.size();
    int l = 0, r = n;
    while (l < r)
    {
        int mid = (l + r) / 2;
        if (a[mid] < target) l = mid + 1;
        else                 r = mid;
    }
    return l;
}
```

這個不知道會跑多少次

這兩個狀況都會把陣列從中間切開，取某一邊

每次都（大概）把搜尋範圍砍一半，總時間複雜度 $O(\log_2(n))$

算算複雜度

小心不要複製，會變 $\Theta(n)$！

```
using namespace std;
long long binary_search(const vector<int> &a /* 非嚴格遞增 */,
                        int target)
{
    int n = (int)a.size();
    int l = 0, r = n;
    while (l < r)
    {
        int mid = (l + r) / 2;
        if (a[mid] < target) l = mid + 1;
        else                 r = mid;
    }
    return l;
}
```

這個不知道會跑多少次

這兩個狀況都會把陣列從中間切開，取某一邊

每次都（大概）把搜尋範圍砍一半，總時間複雜度 $O(\log_2(n))$

算算複雜度

```
using namespace std;
void sort(vector<int> &arr)
{
    std::sort(arr.begin(), arr.end());
}
```

Sprout

算算複雜度

```
using namespace std;
void sort(vector<int> &arr)
{
    std::sort(arr.begin(), arr.end());
}
```

→ 這個不知道會跑多久

Sprout

算算複雜度

```
using namespace std;
void sort(vector<int> &arr)
{
    std::sort(arr.begin(), arr.end());
}
```

→ 這個不知道會跑多久

Standard Library 通常會在他的 Documentation 告訴你！

Complexity

Given N as `last - first`:

1,2) $O(N \cdot \log(N))$ comparisons using `operator<`(until C++20) `std::less{}` (since C++20).

3,4) $O(N \cdot \log(N))$ applications of the comparator `comp`.

https://en.cppreference.com/w/cpp/algorithm/sort

算算複雜度

輾轉相除法

```
int euclid_gcd(int a, int b)
{
    return b == 0 ? a : euclid_gcd(b, a % b);
}
```

算算複雜度

輾轉相除法

```
int euclid_gcd(int a, int b)
{
    return b == 0 ? a : euclid_gcd(b, a % b);
}
```

→ 這個不知道會遞迴幾層

算算複雜度

輾轉相除法

```
int euclid_gcd(int a, int b)
{
    return b == 0 ? a : euclid_gcd(b, a % b);
}
```

→ 這個不知道會遞迴幾層

- （不失一般性假設 a, b 都是正的）
- 如果 $b > a$，那下次呼叫 a, b 會對換

算算複雜度

輾轉相除法

```
int euclid_gcd(int a, int b)
{
    return b == 0 ? a : euclid_gcd(b, a % b);
}
```

→ 這個不知道會遞迴幾層

- （不失一般性假設 a, b 都是正的）
- 如果 $b > a$，那下次呼叫 a, b 會對換
- 如果 $b \leq a$，a 除以 b 的餘數會有這樣的性質：

$$a = qb + r, 0 \leq r < b \Longrightarrow a = qb + r \geq b + r > 2r$$

每次餘數都會至少砍半！

算算複雜度

輾轉相除法

```
int euclid_gcd(int a, int b)
{
    return b == 0 ? a : euclid_gcd(b, a % b);
}
```

→ 這個不知道會遞迴幾層

- （不失一般性假設 a, b 都是正的）
- 如果 $b > a$，那下次呼叫 a, b 會對換
- 如果 $b \leq a$，a 除以 b 的餘數會有這樣的性質：

$$a = qb + r, 0 \leq r < b \Longrightarrow a = qb + r \geq b + r > 2r$$

  每次餘數都會至少砍半！
- 時間複雜度是 $O(\log \min(a, b))$

算算複雜度

輾轉相除法

```
int euclid_gcd(int a, int b)
{
    return b == 0 ? a : euclid_gcd(b, a % b);
}
```

→ 這個不知道會遞迴幾層

- （不失一般性假設 a, b 都是正的）
- 如果 $b > a$，那下次呼叫 a, b 會對換
- 如果 $b \leq a$，a 除以 b 的餘數會有這樣的性質：

$$a = qb + r, 0 \leq r < b \Longrightarrow a = qb + r \geq b + r > 2r$$

  每次餘數都會至少砍半！
- 時間複雜度是 $O(\log \min(a, b))$
- 什麼時候會真的要跑這麼久？

算算複雜度

二進位加法

```
void enumerate_binary(int n)
{
    std::string s(n, '0');
    while (true)
    {
        int i = 0;
        for (; i < n && s[i] == '1'; i++)
            s[i] = '0';
        if (i == n) break;
        s[i] = '1';
    }
}
```

算算複雜度

二進位加法

```
void enumerate_binary(int n)
{
    std::string s(n, '0');
    while (true)
    {
        int i = 0;
        for (; i < n && s[i] == '1'; i++)
            s[i] = '0';
        if (i == n) break;
        s[i] = '1';
    }
}
```

$O(n)$

$O(2^n)$

進位最壞會花 $O(n)$

算算複雜度

- 進位真的有這麼糟嗎？

Sprout

算算複雜度

- 進位真的有這麼糟嗎？
- Idea 1: 進位不會很常發生

Sprout

算算複雜度

- 進位真的有這麼糟嗎？

- Idea 1: 進位不會很常發生
- 每 2^{i+1} 個數字第 i 位才會發生進位！

$$\sum_{i=0}^{n-1} \frac{2^n}{2^{i+1}} \leq 2^n$$

算算複雜度

- 進位真的有這麼糟嗎？

- Idea 1: 進位不會很常發生
- 每 2^{i+1} 個數字第 i 位才會發生進位！

$$\sum_{i=0}^{n-1} \frac{2^n}{2^{i+1}} \leq 2^n$$

- 所以爬到進位全部只花 $O(2^n)$，真的進那一位每次花 $O(1)$，總共是 $O(2^n)$

算算複雜度

- 進位真的有這麼糟嗎？

算算複雜度

- 進位真的有這麼糟嗎？
- Idea 2: 進位發生的時候兩個 1 會變一個 1

Sprout

算算複雜度

- 進位真的有這麼糟嗎？

- Idea 2: 進位發生的時候兩個 1 會變一個 1
- 放一個 1 進去的時候付兩塊錢

Sprout

算算複雜度

- 進位真的有這麼糟嗎？

- Idea 2: 進位發生的時候兩個 1 會變一個 1
- 放一個 1 進去的時候付兩塊錢
- 第一塊錢用來付一單位「把 1 放進去」的時間

Sprout

算算複雜度

- 進位真的有這麼糟嗎？

- Idea 2: 進位發生的時候兩個 1 會變一個 1
- 放一個 1 進去的時候付兩塊錢
- 第一塊錢用來付一單位「把 1 放進去」的時間
- 第二塊錢用來付一單位「被進位」的時間

算算複雜度

- 進位真的有這麼糟嗎？

- Idea 2: 進位發生的時候兩個 1 會變一個 1
- 放一個 1 進去的時候付兩塊錢
- 第一塊錢用來付一單位「把 1 放進去」的時間
- 第二塊錢用來付一單位「被進位」的時間
- 所以所有的時間都可以被這些錢付完，總共是 $O(2^n)$

算算複雜度

- 枚舉前 2^n 個數字每次平均起來每次花 $O(1)$
- 均攤 $O(1)$
- 均攤分析

現實因素

複雜度以外的因素

複雜度比較好的，程式一定跑得比較快嗎？

Sprout

複雜度以外的因素

複雜度比較好的，程式一定跑得比較快嗎？

- 複雜度只有保證「n 足夠大的時候」

複雜度以外的因素

複雜度比較好的，程式一定跑得比較快嗎？

- 複雜度只有保證「n 足夠大的時候」
  - 很快的 $O(n \log n)$ 跟有一些大常數的 $O(n)$

複雜度以外的因素

複雜度比較好的，程式一定跑得比較快嗎？

- 複雜度只有保證「n 足夠大的時候」
  - 很快的 $O(n \log n)$ 跟有一些大常數的 $O(n)$
- 就算是同樣的複雜度，也有常數大小之分

Sprout

算算複雜度

同樣假設是一單位時間的運算，如果 word size 是 w

- 加減乘除法的電路深度都是 $\Theta(\log w)$！

Sprout

算算複雜度

同樣假設是一單位時間的運算，如果 word size 是 w

- 加減乘除法的電路深度都是 $\Theta(\log w)$！
  - 加就是減
  - 乘法電路比較深
  - 除法通常是用常數比例大小的乘法湊出來

Sprout

算算複雜度

同樣假設是一單位時間的運算，如果 word size 是 w

- 加減乘除法的電路深度都是 $\Theta(\log w)$！
  - 加就是減
  - 乘法電路比較深
  - 除法通常是用常數比例大小的乘法湊出來
- 記憶體存取

Sprout

算算複雜度

同樣假設是一單位時間的運算，如果 word size 是 w

- 加減乘除法的電路深度都是 $\Theta(\log w)$！
  - 加就是減
  - 乘法電路比較深
  - 除法通常是用常數比例大小的乘法湊出來
- 記憶體存取
  - 一直要連續的記憶體很容易預測
  - 亂跳一通會 cache miss

算算複雜度

- 演算法的世界
- 程式競賽的世界

Sprout

算算複雜度

- 演算法的世界
  - 複雜度比較好就是更好！
- 程式競賽的世界

Sprout

算算複雜度

- 演算法的世界
  - 複雜度比較好就是更好！
- 程式競賽的世界
  - 通常有一個時間限制
  - 比較鼓勵大家想出複雜度好的做法

Sprout

算算複雜度

- 演算法的世界
  - 複雜度比較好就是更好！
- 程式競賽的世界
  - 通常有一個時間限制
  - 比較鼓勵大家想出複雜度好的做法
- 現實世界

Sprout

算算複雜度

- 演算法的世界
  - 複雜度比較好就是更好！
- 程式競賽的世界
  - 通常有一個時間限制
  - 比較鼓勵大家想出複雜度好的做法
- 現實世界
  - 常數小一點，壓多少就是快多少！
  - 複雜度低，後面常數輸掉可能就不好用了
  - 開發維護成本

Sprout

P vs NP

Sprout

計算理論

- 整天聽到大家問 $P \stackrel{?}{=} NP$，但這是什麼意思？

Sprout

計算理論

- 圖靈機
- 用數學語言抽象描述的電腦
- 基本上跟 word RAM 電腦（多項式以內）一樣強

Sprout

計算理論

計算理論想要回答「我們的電腦有多強？」

- 有電腦解決不了的問題！（停機問題）

Sprout

計算理論

計算理論裡面的「問題」是決定性問題

- 輸入幫你擺在機器裡面
- 要花多久的時間才能決定 Yes / No？

Sprout

計算理論

計算理論裡面的「問題」是決定性問題

- 輸入幫你擺在機器裡面
- 要花多久的時間才能決定 Yes / No？
- 例子：給你正整數 a, b, c，是不是 $a \times b = c$？

計算理論

P 問題

- 可以在多項式時間內決定 Yes / No

Sprout

計算理論

P 問題

- 可以在多項式時間內決定 Yes / No
- 例子 1：給你一張圖 G，問他是不是一棵樹

Sprout

P 問題

- 可以在多項式時間內決定 Yes / No
- 例子 1：給你一張圖 G，問他是不是一棵樹
- 例子 2：給你一個正整數 n，問他是不是一個質數

計算理論

例子 2：給你一個正整數 n，問他是不是一個質數

- 直接除除看 n 是不是 $1 \sim n$ 的因數，這樣應該是多項式時間？

計算理論

例子 2：給你一個正整數 n，問他是不是一個質數

- 直接除除看 n 是不是 $1 \sim n$ 的因數，這樣**不是**多項式時間

例子 2：給你一個正整數 n，問他是不是一個質數

- 直接除除看 n 是不是 $1 \sim n$ 的因數，這樣**不是**多項式時間
- 輸入的 n 是用進位方法表達的，所以大小只有 $\Theta(\log n)$！
  這個前提下的「多項式」是 $\log n$ 的多項式

例子 2：給你一個正整數 n，問他是不是一個質數

- 直接除除看 n 是不是 $1 \sim n$ 的因數，這樣**不是**多項式時間
- 輸入的 n 是用進位方法表達的，所以大小只有 $\Theta(\log n)$！
  這個前提下的「多項式」是 $\log n$ 的多項式
- 其實是可以做的，但是太困難了

計算理論

NP 問題

- 可以在**非確定性**多項式時間內決定 Yes / No
- 非確定性可以猜，但條件是 Yes 要有辦法猜對，No 怎麼樣都不可以猜對
- 所有的 P 問題一定是 NP 問題
- 可以在多項式時間內驗證 Yes / No
- 也就是說 Yes 的輸入都可以找到一個多項式大小的**證據**，但 No 的都找不到

NP 問題

- 漢米爾頓路徑：給定一個圖 G，有沒有一條簡單路徑通過所有點？
  - 非確定性：直接一直猜下一個點
  - 多項式驗證：證據就是那條路徑
- 子集合和：給你一些物品的體積 $v_1, \ldots, v_n$，可不可以剛剛好塞體積 $= V$？
  - 非確定性：直接一直猜每個東西要不要裝
  - 多項式驗證：證據就是要怎麼裝

計算理論

NP-Hard 問題

- 任意 NP 問題的輸入，都可以在多項式內被「計算」成這個問題的輸入，使得答案是一樣的

NP-Complete 問題

- 既是 NP 問題也是 NP-Hard
- 所以只要任何一個 NP-Complete 問題被證明在或不在 P 裡面，NP 就會全部都在或不在裡面！

計算理論

$$P \stackrel{?}{=} NP$$

- 普遍大家「相信」是 $\neq$
- 不過沒有人能證明，所以各種 NP 問題難有快速的作法
- 不小心發現某個問題可以解決一個 NP-Complete 問題，不是很難就是做不好！

計算理論

事實上，

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$$

- L 是確定性對數「空間」，時間任意（不過不可能超過多項式）
- NL 是非確定性對數「空間」
- PSPACE 是確定性多項式「空間」，時間任意

事實上，

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$$

目前只知道這是 $\neq$

- L 是確定性對數「空間」，時間任意（不過不可能超過多項式）
- NL 是非確定性對數「空間」
- PSPACE 是確定性多項式「空間」，時間任意

事實上，

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$$

目前只知道這是 $\neq$

- L 是確定性對數「空間」，時間任意（不過不可能超過多項式）
- NL 是非確定性對數「空間」
- PSPACE 是確定性多項式「空間」，時間任意
- 還有更多問題類別！