



Complexity

上課補充 by PixelCat

Sprout



課程影片

Sprout



課程影片

Q & A ?

Sprout



數學的複雜度定義

Sprout



極限？

課程影片說複雜度是用極限去算的

Definition 極限

$\lim_{n \rightarrow \infty} f(n) = L$ ，定義為：對任意的 $\epsilon > 0$ ，存在某個 N ，使得對於所有 $n > N$ ， $|f(n) - L| < \epsilon$ 。

Sprout



極限？

課程影片說複雜度是用極限去算的

Definition 極限（白話版本）

$\lim_{n \rightarrow \infty} f(n) = L$ ，表示在 n 超大的時候 $f(n)$ 會很接近很接近某個確切的值

有時候極限會不存在： $f(n)$ 可以無限制的變大、 $f(n)$ 沒有定義、等等情況

Sprout



漸近複雜度

複雜度的定義和極限長很像（但不太一樣）

Definition Big-O 複雜度

$f(n) = O(g(n))$ ，定義為：存在常數 $N, c > 0$ ，使得對於所有 $n \geq N$ ，
 $0 \leq f(n) \leq cg(n)$ 。

Sprout



漸近複雜度

複雜度的定義和極限長很像（但不太一樣）

Definition Big-O 複雜度

$f(n) = O(g(n))$ ，表示在 n 超大的時候， $f(n)$ 和 $g(n)$ 的常數倍比大小， $f(n)$ 會一直輸下去。

Sprout



漸近複雜度

Corollary 推論 (錯誤)

$f(n) = O(g(n))$ ，代表： $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq \infty$

這是錯的！

Sprout



漸近複雜度

Corollary 推論（正確）

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \neq \infty, \text{ 代表: } f(n) = O(g(n))$$

兩者有微妙的小差異

Sprout



漸近複雜度

複雜度不在乎你的函數具體是多少，只想比較兩個函數在遙遠的未來誰會比較大

我們不在乎（沒辦法在乎）演算法具體要跑多久，只想知道兩個演算法在超大規模下誰需要比較少算力（比較快）

Sprout



漸近複雜度

small-O	$f(n) = o(g(n))$	$\dots f(n) < cg(n) \dots$
big-O	$f(n) = O(g(n))$	$\dots f(n) \leq cg(n) \dots$
big-theta	$f(n) = \Theta(g(n))$	$\dots c_1g(n) \leq f(n) \leq c_2g(n) \dots$
big-omega	$f(n) = \Omega(g(n))$	$\dots cg(n) \leq f(n) \dots$
small-omega	$f(n) = \omega(g(n))$	$\dots cg(n) < f(n) \dots$

Sprout



漸近複雜度

small-O	$f(n) = o(g(n))$	$g(n)$ 是 $f(n)$ 的 上界 (嚴格大於)
big-O	$f(n) = O(g(n))$	$g(n)$ 是 $f(n)$ 的 上界 (可以一樣)
big-theta	$f(n) = \Theta(g(n))$	$f(n)$ 跟 $g(n)$ 長一樣快
big-omega	$f(n) = \Omega(g(n))$	$g(n)$ 是 $f(n)$ 的 下界 (可以一樣)
small-omega	$f(n) = \omega(g(n))$	$g(n)$ 是 $f(n)$ 的 下界 (嚴格小於)

Sprout



常見的複雜度與 NP-completeness

Sprout



比一比

誰比較大？

$3n^2 + n + 20$	vs	$100n$
n^{100}	vs	2^n
n^2	vs	$10n \log n$
100^n	vs	$n!$
30×2^n	vs	3^n
$100n$	vs	$200n$

(credit：古代投影片 by Chin-Huang Lin)

Sprout



比一比

誰比較大？

(複雜度比較高)	$3n^2 + n + 20$	vs	$100n$	
	n^{100}	vs	2^n	(複雜度比較高)
(複雜度比較高)	n^2	vs	$10n \log n$	
	100^n	vs	$n!$	(複雜度比較高)
	30×2^n	vs	3^n	(複雜度比較高)
(複雜度相同)	$100n$	vs	$200n$	(複雜度相同)

(credit：古代投影片 by Chin-Huang Lin)

Sprout



複雜度的階級

誰比較大？

$$\overline{n^2 \quad \text{vs} \quad 2^n}$$

Sprout



複雜度的階級

誰比較大？

n^2	vs	2^n
n^{10}	vs	2^n

Sprout



複雜度的階級

誰比較大？

n^2	vs	2^n
n^{10}	vs	2^n
n^{10}	vs	1.1^n

Sprout



複雜度的階級

誰比較大？

n^2	vs	2^n
n^{10}	vs	2^n
n^{10}	vs	1.1^n
$n^{10000000000}$	vs	1.0000000001^n

Sprout



複雜度的階級

誰比較大？

n^2	vs	2^n
n^{10}	vs	2^n
n^{10}	vs	1.1^n
$n^{1000000000}$	vs	1.0000000001^n

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

Sprout



複雜度的階級

誰比較大？

n^2	vs	$\log n$
n^2	vs	$\log^{10} n$
$n^{0.1}$	vs	$\log^{10} n$
$n^{1.0000000001}$	vs	$\log^{1000000000} n$

Sprout



複雜度的階級

誰比較大？

n^2	vs	$\log n$
n^2	vs	$\log^{10} n$
$n^{0.1}$	vs	$\log^{10} n$
$n^{1.0000000001}$	vs	$\log^{1000000000} n$

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

Sprout



一個問題有多難？

相比於指數時間的演算法，多項式時間是巨大的進步

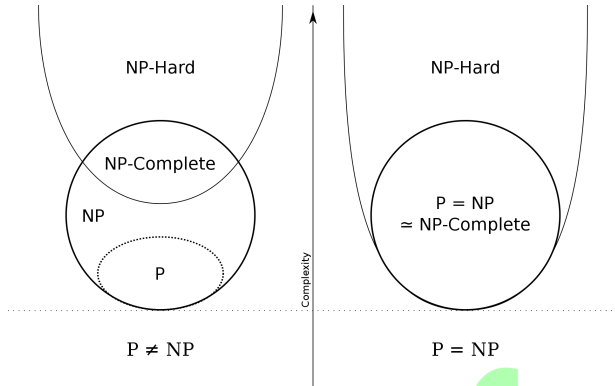
有些問題可以輕鬆設計出多項式時間的演算法

有些問題怎麼努力想，就是只想得到指數時間的演算法

Sprout



NP-completeness



<https://en.wikipedia.org/wiki/NP-completeness>

Sprout



NP-completeness

決定性問題：只能回答 YES/NO 的問題

根據「問題有多難在多項式時間解決」，我們把所有決定性問題歸類：

- P 問題：可以在多項式時間內解決
- NP 問題：可以在多項式時間內**驗證一組 YES 的解確實是對的**
- NP-hard 問題
 - 本身未必是 NP 問題
 - 只要多項式時間做出 NP-hard 問題，就可以多項式時間做出所有 NP 問題
- NP-complete 問題：同時是 NP 問題和 NP-hard 問題

Sprout



NP-completeness

NP 問題能夠在多項式時間解決嗎？

知道答案的話不要告訴我，去發論文你就變世界偉人

目前的普遍信念是 $P \neq NP$

也就是 NP 好難好難，難到沒辦法在多項式時間內解決

Sprout



NP-completeness

「知道問題不可做」和「知道問題要怎麼做」一樣重要

情境一：題目怎麼看起來很像 NP-complete 問題？
→ 可能看錯題目了、漏看條件了、出題者完蛋了

情境二：我怎麼不小心做出 NP-complete 問題了？
→ 想法出錯了

Sprout



分析複雜度

Sprout



分析複雜度

假設某些**基本操作**需要的時間都差不多

1. 計算演算法需要做幾次**基本操作**
2. 留下複雜度最大的那一項

計算複雜度相當仰賴 case by case 討論，不只是數迴圈！

Sprout



分析複雜度：數迴圈（一）

```
#define rep(i, n) for(int i = 0; i < n; i++)
```

```
void mult(int n, int a[N][N], int b[N][N], int c[N][N]) {  
    rep(i, n) rep(j, n) {  
        c[i][j] = 0;  
    }  
    rep(i, n) rep(j, n) rep(k, n) {  
        tmp[i][j] += (a[i][k] * b[k][j]);  
    }  
    rep(i, n) rep(j, n) {  
        c[i][j] = tmp[i][j] % MOD;  
    }  
}
```

Sprout



分析複雜度：數迴圈（一）

- $N^3 + 2N^2$ 次賦值
- N^3 次加法
- N^3 次乘法
- N^2 次除法（模運算）
- ??? 次陣列取值、指標和位址計算...

確切不知道，大概是 $? \times N^3 + ? \times N^2 + \dots$ 次基本操作

Sprout



分析複雜度：數迴圈（一）

- $N^3 + 2N^2$ 次賦值
- N^3 次加法
- N^3 次乘法
- N^2 次除法（模運算）
- ??? 次陣列取值、指標和位址計算...

確切不知道，大概是 $? \times N^3 + ? \times N^2 + \dots$ 次基本操作

複雜度告訴你， N 很大的時候常數和複雜度小的項沒什麼大影響

複雜度就是妥妥的 $O(N^3)$

Sprout



分析複雜度：數迴圈（一）

一層迴圈跑很多次
迴圈套起來迭代次數會乘起來

最多有幾層迴圈，複雜度就是幾次方（?!）

Sprout



分析複雜度：數迴圈（二）

```
void f(int n) {  
    int ans = 0;  
    for(int i = 0; i < n; i++) {  
        for(int j = 0; j < (1 << n); j++) {  
            ans += i * j;  
        }  
    }  
}
```

Sprout



分析複雜度：數迴圈（二）

```
int f(int n) {  
    int ans = 0;  
    for(int i = 0; i < n; i++) { // 0 ... (n - 1)  
        for(int j = 0; j < (1 << n); j++) { // 0 ... (2^n - 1)  
            ans += i * j;  
        }  
    }  
    return ans;  
}
```

時間複雜度： $O(n2^n)$

Sprout



分析複雜度：數迴圈（三）

```
int g() {  
    int ans = 0;  
    for(int i = 0; i < 100; i++) {  
        ans += i;  
    }  
    return ans;  
}
```

Sprout



分析複雜度：數迴圈（三）

```
int g() {  
    int ans = 0;  
    for(int i = 0; i < 100; i++) {  
        ans += i;  
    }  
    return ans;  
}
```

雖然有點不甘願，但是 $O(100) = O(1)$ 確實是常數時間

Sprout



分析複雜度：數迴圈（四）

```
int my_lower_bound(int n, int key, int arr[]) {  
    int lo = -1, hi = n;  
    while(hi - lo > 1) {  
        int mi = (hi + lo) / 2;  
        if(arr[mi] >= key) hi = mi;  
        else lo = mi;  
    }  
    return hi;  
}
```

Sprout



分析複雜度：數迴圈（四）

```
int my_lower_bound(int n, int key, int arr[]) {  
    int lo = -1, hi = n;  
    while(hi - lo > 1) {  
        int mi = (hi + lo) / 2;  
        if(arr[mi] >= key) hi = mi;  
        else lo = mi;  
    }  
    return hi;  
}
```

$(hi - lo)$ 一開始是 $n + 1$ ，每次都被砍一半，砍個 $\log N$ 次之後變 1 退出迴圈

二分搜尋，時間複雜度 $O(\log N)$

Sprout



分析複雜度：被藏起來的複雜度

```
std::sort(a + 1, a + n + 1);
```

沒有迴圈，總共 $O(1)$ (??)

Sprout



分析複雜度：被藏起來的複雜度

```
std::sort(a + 1, a + n + 1);
```

沒有迴圈，總共 $O(1)$ (??)

呼叫別的函數當然需要時間，[cppreference](#) 之類的通常會告訴你各個內建函數的時間複雜度

Sprout



分析複雜度：遞迴函數

```
int gcd(int a, int b) {  
    if(b == 0) return a;  
    return gcd(b, a % b);  
}
```

Sprout



分析複雜度：遞迴函數

```
int gcd(int a, int b) {  
    if(b == 0) return a;  
    return gcd(b, a % b);  
}
```

輾轉相除的時間複雜度是多少？時間複雜度 $O(\log \min(a, b))$

為什麼？！

Sprout



分析複雜度：遞迴函數

遞迴函數比較難搞，在這裡先略過

Sprout



分析複雜度：均攤分析

```
for(int idx = 0; idx < n; idx++) {  
    while(stk.size() && value[stk.top()] > value[idx]) {  
        ans[stk.top()] = idx;  
        stk.pop();  
    }  
    stk.push(idx);  
}
```

上週教過的單調堆疊

Sprout



分析複雜度：均攤分析

```
for(int idx = 0; idx < n; idx++) {  
    while(stk.size() && value[stk.top()] > value[idx]) {  
        ans[stk.top()] = idx;  
        stk.pop();  
    }  
    stk.push(idx);  
}
```

for 迴圈執行 N 次

while 迴圈每一輪最多執行 N 次 (stack 最多裝 N 個元素)

總複雜度是 $O(N^2)$ ，真的那麼糟糕嗎

Sprout



分析複雜度：均攤分析

```
for(int idx = 0; idx < n; idx++) {  
    while(stk.size() && value[stk.top()] > value[idx]) {  
        ans[stk.top()] = idx;  
        stk.pop();  
    }  
    stk.push(idx);  
}
```

認真聽上週課程的你知道，不會有那麼多元素讓你 pop，從頭到尾 while 迴圈總共最多跑 N 次。時間複雜度 $O(N)$

均攤分析「偶爾會跑很慢，但是不可能每次都跑很慢，平均起來還是跑很快」





分析複雜度

複雜度分析不單純是數迴圈、還需要豐富的經驗和數學和數學和數學

Sprout



分析複雜度

算完複雜度之後呢？

- 把題目給的變數範圍代進去，看看會不會超時
 - 我的電腦可以一秒跑 4×10^9 次加法
 - 綜合考量其他因素，代入複雜度後在 $10^7 \sim 10^8$ 通常算合理不超時範圍
- 有沒有複雜度差、但夠快而且好寫的作法？
- 超時了，優化演算法的哪個地方可以改進複雜度？
 - 例：少用一層迴圈？

Sprout



複雜度之外的現實因素

Sprout



「常數」

複雜度的計算會忽略常數

在線上評測系統，你不只要考慮演算法的複雜度，還要把他實做出來

- N 次加法和 $2N$ 次加法，誰比較快？
- N 次加法和 N 次除法，誰比較快？
- ... ?

Sprout



實驗一

```
const int MAXN = 100'000'000;  
int a[MAXN + 10]; // is assigned random value  
  
for(int i = 1; i <= MAXN; i++) ans = ans ^ a[i];  
// 0.021 s  
  
for(int i = 1; i <= MAXN; i++) ans = ans + a[i];  
// 0.022 s  
  
for(int i = 1; i <= MAXN; i++) ans = ans * a[i];  
// 0.065 s  
  
for(int i = 1; i <= MAXN; i++) ans = (ans * a[i]) % MOD;  
// 0.292 s
```

Sprout



實驗二

```
const int MAXN = 100'000'000;  
int a[MAXN + 10];    // is assigned random value in [0, 2^16)  
int ord[MAXN + 10];  // is assigned 1 ... MAXN  
  
for(int i = 1; i <= MAXN; i++) ans += a[ord[i]];  
// 0.035 s  
  
shuffle(ord + 1, ord + MAXN + 1);  
for(int i = 1; i <= MAXN; i++) ans += a[ord[i]];  
// 0.758 s
```

「cache miss」

Sprout



實驗三

```
const int MAXN = 10'000;  
int a[MAXN + 10][MAXN + 10]; // is assigned random value in [0, 2^16)  
  
for(int i = 1; i <= MAXN; i++)  
    for(int j = 1; j <= MAXN; j++)  
        ans += a[i][j];  
// 0.085 s  
  
for(int j = 1; j <= MAXN; j++)  
    for(int i = 1; i <= MAXN; i++)  
        ans += a[i][j];  
// 1.741 s
```

也是 cache miss

Sprout



實驗四

```
const int MAXN = 100'000'000;  
int a[MAXN + 10]; // is assigned random value in [0, 2^16)  
  
for(int i = 1; i <= MAXN; i++) ans = ans + a[i];  
// g++ main.cpp -O0  
// 0.166 s  
  
for(int i = 1; i <= MAXN; i++) ans = ans + a[i];  
// g++ main.cpp -O4  
// 0.045 s
```

編譯器的邪惡優化

Sprout



常數優化

除了演算法以外，還有無數種因素會影響你的程式跑多久

「常數優化」的目標是透過各種（邪惡的）手段，寫出演算法相同但執行更快的程式碼

Sprout



常數優化

但是這些手段通常優化效果有限

「複雜度壓一個 N ， $N = 1000$ 」和「常數優化讓程式變快 20%」哪一個比較賺？

設計更好的演算法更加重要！

Sprout