競技程式設計導論

Introduction to Competitive Programming

2023 南 11 校聯合寒訓 - 演算法課程

2023.01.30

陳俊安 Colten

主辦單位：

協辦單位：

贊助單位：

本次寒訓演算法講師陣容

- 成功大學資訊工程系 大一 陳俊安 Colten
- 成功大學資訊工程系 大一 洪睿廷 Eric
- 成功大學資訊工程系 大一 劉哲佑 Jason
- 嘉義高中 高三 陳克盈 Koying

陳俊安 Colten

- 國立成功大學資訊工程系 大一 (特選甲組)
- 熱愛競技程式設計、演算法、資料結構
- APCS 實作滿級分
- 台南女中資訊研究社 37th C++ 進階班講師
- 2023 資訊之芽培訓計畫 南區算法班 講師
- 2022 ICPC Asia Taoyuan Regional Programming Contest Bronze Medal
- NHDK Ten Point Round Problem Setter
- NHDK 南北四校聯合資研暑期選手培訓課程 講師

陳克盈 Koying

- 準國立清華大學資訊工程系 (特選)
- 嘉義高中 高三
- APCS 滿級分
- 2022 成大高中生程式設計邀請賽 第九名
- 2023 資訊之芽培訓計畫 南區算法班 講師
- CISC 中學資訊討論群 創辦人
- NHDK Ten Point Round 總召
- NHDK 南北四校聯合資研暑期選手培訓課程 講師

劉哲佑 Jason

- 國立成功大學資訊工程系 大一 (APCS 組)
- 2022 NCPC 全國大專電腦軟體設計競賽 佳作
- 武陵高中資訊科技研究社 25th 社長
- 2022 NCKU GDSC Core Member
- 2021 YTP 少年圖靈計畫 決賽晉級
- 專案！專案！專案！
  - Github: https://github.com/jason810496
  - 無聊的時候可以來**逛逛** ><

洪睿廷 Eric

- 國立成功大學資訊工程系 大一 (特選甲組)
- 2022 NCPC 全國大專電腦軟體設計競賽 佳作
- APCS 滿級分
- CPE 大學程式設計檢定 5 題 (1.4%)
- 2021 資訊學科校內賽 第一名
- 2021 資訊學科能力競賽分區複賽 第一名
- 2022 YTP 少年圖靈計畫 決賽晉級

這個營隊的難度

- 東西涵蓋 APCS 實作 5 級分範圍
- 所以難度不會像暑假的營隊這麼恐怖
- 要在 5 天一次吸收這麼多內容是很困難的
- 因此營隊結束後一定要複習

競技程式設計

- aka 演算法競賽、競技程式、競程
- 一種考驗選手演算法與資料結構能力的競賽
- 也是當前高中最主流的比賽
    - 學科能力競賽、NPSC、YTP 都屬於演算法競賽
    - 相關檢定：APCS、CPE

經典技巧 vs 競賽技巧

- 經典技巧之所以經典就是因為其演算法具有一定的知名度
- 因此許多人都理解這一個演算法
    - Ex. Binary Search、Dijkstra、Floyd-Warshall
    - 而 APCS 的考試範圍基本上都屬於經典技巧

經典技巧 vs 競賽技巧

- 但是經典技巧之所以經典也是因為其演算法要能在自己沒有看過的情況下，靠自己想出來是具有一定程度的難度的
- 但是只要你把這一個技巧學起來，那麼你下次再遇到一樣的問題的時候就能運用自如，這也是競程的特色

經典技巧 vs 競賽技巧

- 而競賽技巧就屬於打競程打得很深入時才會遇到的技巧
  - 線段樹、Fenwick Tree、倍增法
  - 這一次的課程主要圍繞在 APCS 範圍內的經典技巧
  - 因為我們時間不多QQ，對競賽技巧有興趣的建議可以去看看 NHDK 之前的暑訓影片
  - ~~這一次的課程內容會比暑假友善 100 倍~~

練競程的方法 - 刷題

- 刷題可以讓自己對於演算法的運用更加的了解
- 有時候剛學完一個演算法可能懂理念，但是不會運用
- 這個時候刷題就非常非常重要了！

接下來要講的東西

- 為了避免大家還沒有準備好一些先備知識
- 因此貼心的課程組準備了這一堂課讓大家暖暖身
- 後面有許多內容非常需要仰賴前面的先備知識才能理解

一切的起點 - 時間複雜度

- 一個可以用來估計程式效率的指標
- 你可能常常會看到有人說他的時間複雜度是 O(N)、O(NM)
- 這一個 O 的意思其實是表示 "在最壞的情況下" 的意思
- 通常我們打競程所估的時間複雜度都是以最壞情況下為主
- 而 O 括號裡面數字越大，則表示效率越差

為什麼都要以最壞的情況下為主

- 通常一個 "正常" 的題目都會給你題目範圍
- 而一個 "正常" 的出題者會在測試資料中加入一些極限測資
- 因此我們在估計的時候當然也要考慮到最極限的狀況

如果不正常的題目或比賽怎麼辦

- 自己通靈
- 對，真的是這樣
- 如果有裁判的話可以問裁判
- 但通常你不會得到你想要的回覆的
- 競程界有很多相關的傳奇故事，可以去打聽看看XD

1 秒能執行多少次

- 1 秒大約能執行 10^8 次左右
- 估完時間複雜度後把數字代進去看看
- 大家要有一個習慣，設計完一個演算法後都請先估計看看時間複雜度的好壞
- 必須要先對自己的程式執行的效率有一個底

時間複雜度最簡單的估計方式

- 算迴圈執行的次數
- 幾次就是多少
- O(N)

```
 5 int main()
 6 {
 7     int n;
 8     cin >> n;
 9
10     for(int i=0;i<n;i++)
11     {
12         cout << "Hello world!\n";
13     }
14
15     return 0;
16 }
```

時間複雜度最簡單的估計方式

- 算迴圈執行的次數
- 幾次就是多少
- O(NM)

```
int main()
{
    int n;
    cin >> n >> m;

    for(int i=0;i<n;i++)
    {
        for(int k=0;k<m;k++)
        {
            cout << "Hello world!\n";
        }
    }

    return 0;
}
```

時間複雜度最簡單的估計方式

- 算迴圈執行的次數
- 幾次就是多少
- O(logN)

```
 5 int main()
 6 {
 7     int n;
 8     cin >> n;
 9
10     while( n ≠ 0 )
11     {
12         n /= 2;
13     }
14
15     return 0;
16 }
```

時間複雜度最簡單的估計方式

- 通常我們會把一些不太重要的常數去掉
- O(N)

```
 5 int main()
 6 {
 7     int n;
 8     cin >> n;
 9     for(int i=0;i<n+5;i++)
10     {
11         cout << "Hello world\n";
12     }
13
14     return 0;
15 }
```

時間複雜度最簡單的估計方式

- 基本上沒執行幾次
- O(1)

```
 5 int main()
 6 {
 7     cout << "Colten\n";
 8
 9     return 0;
10 }
```

一些要記得的

- C++ 內建的 std::sort 的時間複雜度是 O(NlogN)
- O(N/1 + N/2 + N/3 + ⋯ + N/N) 約等於 O(NlogN)
    - 這個是調和級數，有興趣的可以參閱維基百科
    - 要證明的話要用 積分 : (

空間複雜度

- 跟時間複雜度一樣，我們通常都探討最壞的情況
- 時間複雜度探討的是效率、而空間複雜度探討的是記憶體的使用量

空間複雜度

- 通常會大量使用到空間的地方是陣列或一些 STL 資料結構
- 最簡單的估計方式是看看這一些東西塞到全滿的狀態會需要多少數字的空間
- 影響空間複雜度的因素其實還有型態 (int, long long, double)

算術運算子

- 就跟大家平時用的數學一樣
- 在程式中也遵守著數學的規則
  - 做除法時，不能除以 0
  - 先乘除，後加減，有括號先算

```cpp
#include <iostream>

using namespace std;

int main()
{
    int a = 10, b = 5;

    cout << a + b << "\n"; // 15
    cout << a - b << "\n"; // 5
    cout << a * b << "\n"; // 50
    cout << a / b << "\n"; // 2
    cout << a + b * a << "\n"; // 60
    cout << b * ( a + b ) << "\n"; // 75
}
```

算術運算子 %

- 取餘數

```cpp
#include <iostream>

using namespace std;

int main()
{
    int a = 10, b = 7;

    cout << a % b << "\n"; // 3
}
```

算術運算子

- 在做運算的時候要特別小心型態的問題
- 請見下方程式碼
- Why?

```cpp
#include <iostream>

using namespace std;

int main()
{
    int a = 10, b = 3;

    double a2 = 10.0;

    cout << a / b << "\n"; // 3
    cout << a2 / b << "\n"; // 3.33333
}
```

算術運算子

- 由於 a,b 都是整數，而整數跟整數做運算會預設結果出來也是整數
  - a / b = 3.33333 只取整數部分 -> 3
- 由於 a2 是浮點數了， 因此出來的結果為 3.33333

```cpp
#include <iostream>

using namespace std;

int main()
{
    int a = 10, b = 3;

    double a2 = 10.0;

    cout << a / b << "\n"; // 3
    cout << a2 / b << "\n"; // 3.33333
}
```

算術運算子

- 還有一個常犯的錯誤
  - 如果 a,b 都是 int
  - 當 a + b 會超出 int 範圍且沒有使用 long long 儲存的情況下會發生 overflow (溢位) 的問題，造成出來的結果不如預期

比較運算子

- 如果在程式中使用比較運算子做運算，得到的結果會是 布林 的型態
  - 布林 bool
- 當敘述成立時得到的結果為 true (1)，反之為 false (0)
- 常見比較運算子 (<, <=, >, >=, !=, ==)
  - ! 通常我們會稱之為 NOT，因此 != 就是 不等於 的意思
  - == 是指相等的意思，是兩個等於！
  - 用括號包起來避免衝突

```
int a = 10 , b = 5;

cout << ( a > b ) << "\n"; // 1
cout << ( a < b ) << "\n"; // 0
cout << ( a == b ) << "\n"; // 0
```

位元運算子

- 會將數字轉換成 二進位 做完運算後再將結果轉回 十進位 回傳
- 大家應該都對二進位與十進位沒什麼概念，因此這邊我們介紹一下

十進位 與 二進位

- 平時我們在現實生活中都是使用十進位做運算
- 為什麼被稱之為十進位？可以想像成我們做加法的時候都是 10 要進位
- 舉一反三，二進位當然就是 2 的時候要進位
  - 因此十進位的數字會出現 0 ~ 9
  - 而二進位的數字指會出現 0 或 1

十進位轉二進位

- 短除法
  - 將一個數字不斷的使用 2 做短除法，並將餘數紀錄起來
  - 紀錄的餘數倒著回去就會是該數字轉成二進位的結果
  - Example: (因此 12 的二進位是 1100)
    - 12 … 0
    - 6 … 0
    - 3 … 1
    - 1 … 1

二進位轉十進位

- 先想像一個十進位的數字是怎麼被組合出來的
  - 1234
    - 0 * 10 + 1 = 1
    - 1 * 10 + 2 = 12
    - 12 * 10 + 3 = 123
    - 123 * 10 + 4 = 1234

二進位轉十進位的第一種方法

- 那我們把同樣的策略放在二進位上
  - 1011
    - 0 * 2 + 1 = 1
    - 1 * 2 + 0 = 2
    - 2 * 2 + 1 = 5
    - 5 * 2 + 1 = 11
  - 因此 1011 轉成十進位的結果是 11

二進位轉十進位的第二種方法

- 我們一樣先看看十進位的數字
  - 1234
    - $1000 * 1 + 100 * 2 + 10 * 3 + 1 * 4 = 1234$
    - $1000 = 10^3$
    - $100 = 10^2$
    - $10 = 10^1$
    - $1 = 10^0$

二進位轉十進位的第二種方法

- 我們換成二進位的數字
  - 1011
    - $8 * 1 + 4 * 0 + 2 * 1 + 1 * 1 = 11$
    - $8 = 2^3$
    - $4 = 2^2$
    - $2 = 2^1$
    - $1 = 2^0$

AND 運算

- 兩個同位元的數字必須同時為 1，AND 出來的結果才會是 1
  - 1 AND 0 = 0
  - 0 AND 1 = 0
  - 0 AND 0 = 0
  - 1 AND 1 = 1

OR 運算

- 兩個同位元的數字其中一個是 1，OR 出來的結果就會是 1
  - 1 OR 0 = 1
  - 0 OR 1 = 1
  - 0 OR 0 = 0
  - 1 OR 1 = 1

XOR 運算

- 兩個同位元的數字不一樣，XOR 出來的結果就會是 1
  - 1 XOR 0 = 1
  - 0 XOR 1 = 1
  - 0 XOR 0 = 0
  - 1 XOR 1 = 0

實際運算

- 11 XOR 13
  - 先把這兩個數字轉成二進位
    - 1011 (11)
    - 1101 (13)
    - 同位置的位元去做運算
      - 1 XOR 1 = 0
      - 0 XOR 1 = 1
      - 1 XOR 0 = 1
      - 1 XOR 1 = 0
    - 因此 11 XOR 13 的二進位結果為 0110 = 110 轉成十進位就是 6

實際運算

- 小心平時在做位元運算的時候會有 **優先順序** 的問題，為了避免造成不是自己要的結果，我們通常會使用 **括號** 來避免
  - 因為括號的優先權最大，有括號先算

```
5 int main()
6 {
7     int a = ( ( 10 ^ 20 ) & 30 );
8 }
```

補充：向左位移 <<

- 將十進位數字轉成二進位後向左邊後推 k 位，新的位置補 0
- Example：
  - 二進位 1101 向左位移 2 位會變成二進位的 110100

```
 4
 5 int main()
 6 {
 7     int a = 11;
 8
 9     cout << ( a << 2 );
10 }
```

補充：向右位移 >>

- 將十進位數字轉成二進位後向右邊後推 k 位，少掉的部分直接丟掉
- Example：
  - 二進位 1101 向右位移 2 位會變成二進位的 11

```
 5 int main()
 6 {
 7     int a = 2023;
 8
 9     cout << ( a >> 2 );
10 }
```

補充：向左、右位移的一些小祕密

- 我們來活用一下剛學到的二進位觀念
- 由於向左位移 k 位會多出 k 個位置，我們在二進位轉十進位時，每到下一個位置就要 *2，換句話說多 k 位出來我們就要多乘 k 次 2
- 因此藉由這一個想法我們可以得出一個結論
  - 十進位的 a 向左位移 k 位的結果會變成 a * (2 的 k 次方)
- 那麼向右位移 k 位？
  - 十進位的 a 向右位移 k 位的結果會變成 …
  - 留給大家舉一反三當作練習

學完運算子你還必須要知道的事情

- 平常如果我們把一個變數 = 某一個值，這一個動作我們就稱為賦值
  - Example:
    - a = 10
- 程式在做這一個動作時，會先講等於右邊的東西處理完
- 右邊的東西處理完之後再將處理完的結果賦予給等於左邊的變數

```
7     int q = 10 + 20;
8
9     cout << q; // 30
```

現在 q = 10, 我想把 q 的值加上 p

```
 5 int main()
 6 {
 7     int q = 10 , p = 20;
 8
 9     q = q + p; // q + p = 30 , 把 30 的結果賦予給 q
10
11     cout << q << " " << p << "\n"; // q = 30, p = 20
12 }
```

學完運算子你還必須要知道的事情

- 平時我們在寫 a = a + b 這類的東西時可以簡寫成
  - a += b
- 只要是運算類的符號基本上都可以適用
  - a -= b
  - a *= b
  - a /= b
  - a ^= b
  - a <<= b

```
 5 int main()
 6 {
 7     int a = 2023;
 8
 9     a += 1;
10
11     cout << a; // 2024 Happy New Year!
12 }
```

學完運算子你還必須要知道的事情

- 把某個變數 +1 或 -1 還可以再縮寫成
  - 變數++ 或 變數--

```
 5 int main()
 6 {
 7     int a = 3;
 8
 9     a++;
10     cout << a << "\n"; // 4
11     a--;
12     cout << a << "\n"; // 3
13 }
```

初始化

- 做一些變數運算時記得小心變數初始化的問題
- 就很像那個變數不知道是什麼東西，但你把它拿去做運算
- 就會發生無法預測的事情

```
 5 int main()
 6 {
 7     int q; // q 不知道是什麼
 8
 9     q++; // 不知道 q 是什麼卻把它 +1
10 }
```

前置與後置

- 加加還有減減也可以放在變數的前面，意思是不一樣的
  - 放在後面表示 **後運算**，也就是該做的東西都處理完才做 +1 或 -1
  - 反之則為 **前運算**，也就是先 +1 或 -1 才處理接下來的事情

```
 5 int main()
 6 {
 7     int a = 3;
 8
 9     cout << a++ << "\n"; // 3
10     cout << a << "\n"; // 4
11 }
```

```
 5 int main()
 6 {
 7     int a = 3;
 8
 9     cout << ++a << "\n"; // 4
10     cout << a << "\n"; // 4
11 }
```

競程選手的 code 都好毒哦...

- #include <bits/stdc++.h>
- #define int long long
- ios_base::sync_with_stdio(false);
- cin.tie(0);
- 這些是什麼東西？

競程選手的 code 都好毒哦...

- #include <bits/stdc++.h> 萬用標頭檔
- #define int long long 把 int 定義成 long long
- ios_base::sync_with_stdio(false); I/O 加速
- cin.tie(0); I/O 加速
- 這些都是競程選手為了 "方便" 而使用的東西
- ~~不建議你在寫專案的時候這樣用，會被扁~~

一些 Online Judge 術語的部分

- AC (Accepted) 好耶！你的程式 "可能" 是對的
- WA (Wrong Answer) 哈哈！下去，你的程式 "可能" 是錯的
- TLE (Time Limited Exceed) 你的程式執行超久，快一點la
- RE (Runtime Error) 你的程式突然停了，是怎樣la
- CE (Compile Error) 不是la，不要上傳一個不能執行的東西
- MLE (Memory Limited Exceed) 你程式用太多記憶體了，不要這麼浪費la

一些你我都會常犯錯的東西

- long long 該開記得要開
- 陣列索引要小心，不要不小心用出界

現在考考大家

- 給你一個長度不超過 10^5 的序列，接下來查詢 10^5 次某一個區間 [L,R] 的和，請問你會怎麼做
  - ~~不要跟我講線段樹或 BIT，我會請你出去~~

前綴和求區間和

- 暴力顯然複雜度不好，這一題使用到一個超級常見的技巧
- ~~線段樹！！~~
- 哦不，是前綴和，上面有人在亂

前綴和求區間和

- 定義另一個陣列 b_i = a_1 + ⋯ + a_i
- 先做好這一個陣列之後，接下來每一次查詢 [L,R]
- 答案就會是 b_R - b_{L-1}

```cpp
int main()
{
    int n;
    cin >> n;

    for(int i=1;i≤n;i++) cin >> a[i];
    for(int i=1;i≤n;i++) b[i] = b[i-1] + a[i];
    int q;
    cin >> q;
    while(q--)
    {
        int l,r;
        cin >> l >> r;
        cout << b[r] - b[l-1] << "\n";
    }

    return 0;
}
```

二維前綴和

- 我留給 DP 講師了，我不用畫圖了，好爽
- 我覺得二維前綴和的概念蠻適合放在 DP 的
- 所以就交給 DP 講師發揮了

一些你營隊期間可能會使用到的數學

- 指數、對數
- 數學歸納法
- 反證法
- 交換律、結合律、分配律
- 遞迴
- ~~三角函數~~

快速的求 a 的 n 次方

- 單純用迴圈的話時間複雜度會是 O(n)
- 但是其實是有辦法用 分治 的概念去處理成 O(log n) 的
    - 大事化小、小事化無

快速的求 a 的 n 次方

- 接下來要教的這一個技巧我們一般稱 快速冪
- 我們利用 遞迴 求 a 的 n 次方
- 如果 n 是偶數
- 我們可以換成 (a 的 n/2 次方) * (a 的 n/2 次方)
- 計算次數瞬間減少了非常多次！

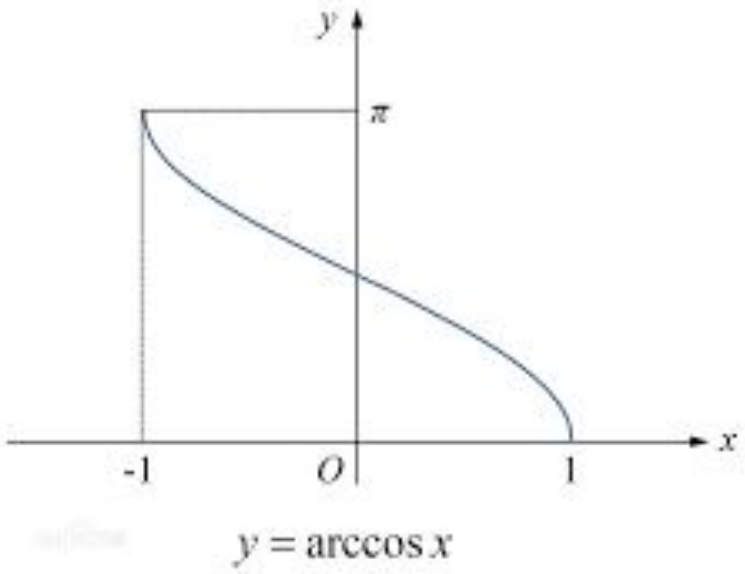
快速的求 a 的 n 次方

- 因此我們在遞迴求的時候當要求的冪次是偶數，都這樣拆解，就能有效的減少非常多不必要的計算

```
 6
 7 int fastpow(int a,int n) // 求 a 的 n 次方
 8 {
 9     if( n == 0 ) return 1; // a^0 = 1
10     else if( n % 2 == 1 ) return a * fastpow(a,n-1); // a^n = a * a^(n-1)
11     else // n 是偶數，優化
12     {
13         int tmp = fastpow(a,n/2); // a^n = (a^(n/2))^2
14
15         return tmp * tmp; // 記得用一個變數存起來再乘，不要呼叫兩次一樣的東西，太浪費了
16     }
17 }
```

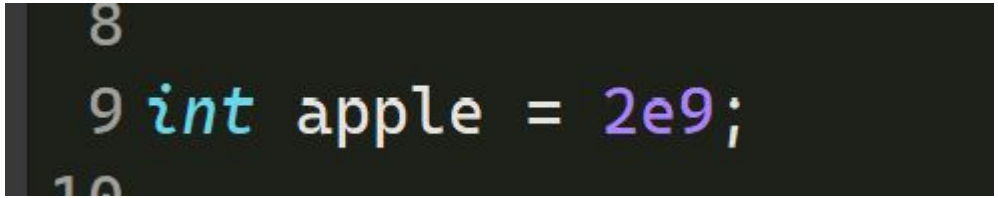
圓周率可以用 acos(-1) 拿到比較準確的數字

- acos 是反餘弦函數



科學記號

- C++ 存在科學記號表示法
- 1e9 = 1 * 10^9
- 2e9 = 2 * 10^9



同餘性質

- 在某些答案數字可能很大的題目都會請你將答案取餘數
- $(a + b + c) \bmod X = (((a + b) \bmod X) + c) \bmod X$
- $(a * b * c) \bmod X = (((a * b) \bmod X) * c) \bmod X$
- $(a - b - c) \bmod X = (((a - b) \bmod X) - c) \bmod X$
- 唯獨除法不滿足此分配律

同餘性質

- 做減法同餘時，當 mod 完數字變負數時要做額外處理
- 必須將這一個負數不斷加 mod 的數字，使其回正

```
 7 int MOD(int num)
 8 {
 9     if( num ≥ 0 ) return num;
10     else
11     {
12         return ( num + ( abs(num) / mod + 1 ) * mod ) % mod;
13     }
14 }
```

const int 的小小優化

- mod 其實是一個常數偏大的操作
- 使用一般的 int 變數儲存 mod 的數字可能會因常數 TLE
- 因此我們會改用 const int

```
7 const int mod = 1e9 + 7;
```

最後教你一些電資圈常用術語

- 這個學得好的話，對你這幾天的社交非常有幫助
- 小心使用，~~被打不要來找我~~

最後教你一些電資圈常用術語

- 電
  - 很強的意思
    - Koying 好電
- 裝弱
  - 字面上的意思, 裝爛
    - Koying 又再裝弱了

最後教你一些電資圈常用術語

- 廚 (Chef)
  - 推捧
    - Koying 整天都在廚人
- 我弱
  - 字面上意思，小心使用
    - Koying 每天都在說我弱，但其實超強

最後教你一些電資圈常用術語

- 這我
  - 字面上意思，這是我
    - Koying: 到底是誰這麼爛阿
    - Colten: 這我
- 裝我
  - 意旨別人愛學你
    - Koying: 我好爛哦
    - Colten: 裝我

最後教你一些電資圈常用術語

- 你好毒哦
  - 原意是真的中毒，但後來延伸出使用太高科技的做法的意思
  - 簡單來說就是 割雞焉用牛刀 的意思
    - Colten: 這題用線段樹好開心
    - Koying: 你好毒哦

最後教你一些電資圈常用術語

- Treap
  - 一種資料結構，能搞定很多事情，也可以放在嘴邊開玩笑
    - Koying: 這一題怎麼做？
    - Colten: Treap
    - Koying: …

最後教你一些電資圈常用術語

- Merge
  - 英文的意思是合併，但延伸出兩個人抱在一起的意思
    - Koying: 我今天 Merge 了 3 個人
    - Colten: 好扯哦，三人同行

Q & A

- aka 聊天時間