TIOJ 終極壓常數大賽

這是在 TIOJ 更新之後想到可以來辦的比賽，感謝大家的參與。也希望大家玩得開心，可以從其中（以及這份題解中）得到一些莫名其妙的收穫 (?

因為做題解的時候出了一些意外（要知道是什麼請繼續往下看），所以導致這麼慢才弄好。

pA. 字串修改

雖然是 pA，不過這題自己認為是這次比賽裡面最好的一題，有很多有趣的性質可以用來優化。題外話一枚，時限不到一秒且輸入大小達到 $N \leq 1.5 \times 10^9$ 的題目應該是絕無僅有了吧 (?

100 分

這題原題是個很簡單的 DP 或 greedy 題。Greedy 的做法就是從字串的後面做回來，如果當前字元是 0，則看該字元的前一個字元是不是也是 0，如果是的話就做第二種操作，否則做第一種。

DP 的話，假設字串是 $s_0 s_1 \cdots s_{N-1}$，令 $\text{dp}[i][0/1]$ 代表把前 s_0 到 s_{i-1} 全部修改成 0 或 1 的答案。如此可以列式：

$$\text{dp}[i][j] = \min(\text{dp}[i-1][j] + [\![s_{i-1} \neq j]\!], \text{dp}[i-1][j\text{^}1] + 1)$$

就也可以在 $O(N)$ 時間解決了。如果寫成滾動 DP 的話，這個 DP 大概就可以拿到 120 分左右的分數。

後面的所有進一步優化因為記憶體限制的緣故，必須要分段讀入字串，greedy 做法變得不可行。也因此，接下來的做法都是從 DP 做法衍生。

輸入優化

這題既然是自己輸入字串，那就可以做輸入優化囉！具體來說是開一個 buffer，然後直接用 `fread` 或 `read` system call 一塊一塊地讀入字串。另一個比較進階、針對 strict

mode（規則 #10）的優化，就是 Linux 可以用 `fcntl` 的 `F_SETPIPE_SZ` 把 pipe buffer size 開大，也可以稍微加速一些。

另外一個小優化是可以發現字串後面補 1 不影響答案，所以最後讀到 EOF 時可以把字串長度填充到例如 64 的倍數，這就可以方便寫個固定大小的內層迴圈，讓編譯器能更方便做 loop unrolling，也會讓後面的各種優化好做很多。

優化一

首先可以觀察一件事情：$\mathrm{dp}[i][0]$ 和 $\mathrm{dp}[i][1]$ 最多只會差 1，因為把全部為 0 做一次第二種操作就變成全部為 1，反之亦同。所以我們可以改成記錄單一的答案 $x_i = \min(\mathrm{dp}[i][0], \mathrm{dp}[i][1])$，並且維護兩個 boolean 值 $a_i = (\mathrm{dp}[i][0] = x + 1)$、$b_i = (\mathrm{dp}[i][1] = x + 1)$。這樣轉移會變成 $a_i = \neg b_{i-1} \wedge (s_{i-1} = 1)$、$b = \neg a_{i-1} \wedge (s_{i-1} = 0)$、$x_i = x_{i-1} + [\![\neg(a_i \vee b_i)]\!]$，就不再需要比較費時的 min 運算，時間上會稍微快一些。

優化二

既然已經是 $O(N)$、每個字元做的事情似乎也沒辦法再少了，那該怎麼辦呢？我們可以試著一次處理很多個字元。具體來說，我們可以把連續 K 個字元當成一個 bit string，轉換成一個介於 $[0, 2^K)$ 的整數 S，然後想辦法用 S 從 a_i, b_i, x_i 一次轉移到 $a_{i+K}, b_{i+K}, x_{i+K}$。由於 x_i 只會遞增、不影響轉移的判斷，可以對所有 $2^K \times 3$ 種 S, a_i, b_i（a_i, b_i 不可能同時為 1）預處理 $a_{i+K}, b_{i+K}, x_{i+K} - x_i$，這樣 DP 就可以一次轉移 K 格了，是一個極大的常數改進。考慮到 locality 和預處理時間，這個表格不能太大，大概 $8 \le K \le 12$ 都算是合理的範圍。另外可以用 `constexpr` 在編譯時期把表打好，可以節省一丁點時間。這個做法開個 `O3,unroll-loops` 大概就可以拿到 200∼230 分。

優化三

既然現在可以一次轉移 K 格了，現在需要一個一個字元處理的就只剩下把字串轉換成 S 的部分了。所幸，這個功能是有 CPU 指令可以直接做到的：在 TIOJ 的環境下（參見 contest 規則的 CPU 資訊），有兩個可以用的函數：BMI2 指令集擴充的 `_pext_u64`，以及 AVX2 指令集擴充的 `_mm256_movemask_epi8`。前者是把某些 bit 位置的 bit 取出來變成一個新的整數，因為 ASCII 0、1 是差在最低的 bit，所以可以例如把字串指標轉換成 `uint64_t` 後用 `_pext_u64(value, 0x0101010101010101)` 把每個 byte 的 lowbit 取出來；後者是把一個 256-bit register 每個 byte 的最高 bit 取出來，所以要用的話還要先用 `_mm256_sll_epi16` 把最低 bit shift 到最高 bit 才能用。後者一次可以一次處理 32 個字元，明顯是比較好的選擇。加了這個優化就可以達到將近 250 分。

重要觀察

接下來這個性質可說是本題最重要的觀察。如果你 K 取的是偶數、把字串轉換成 S 的方法是正常的 Rabin fingerprint（$s_0 s_1 \cdots s_{K-1}$ 轉換成 $s_0 + 2s_1 + 2^2 s_2 + \cdots + 2^{K-1} s_{K-1}$），然後把前一個步驟打的表印出來的話，你會發現一個關於 a_{i+K}, b_{i+K} 很驚人的規律：好像隨著 S 從 0 增加到 $2^K - 1$，一開始 b_{i+K} 是 true、接下來變成兩者都是 false、接下來 a_{i+K} 變成 true。換句話說，存在兩個取決於 a_i, b_i 的常數 C_1, C_2 使得 $b_{i+K} = (S < C_1)$、$a_{i+K} = (S \geq C_2)$！進一步觀察還可以發現 $C_1 = \frac{2^K - 1}{3} + [\![b_i]\!]$、$C_2 = \frac{2 \times 2^K + 1}{3} - [\![a_i]\!]$。

這個性質其實也算好證明，用 $K = 2$ 當作 base case 做數歸就可以了。同時它也很符合直覺：可以想像如果 DP 到最後有連續 0，那就需要多一次操作把它翻成全部是 1，反之亦同。而 C_1, C_2 的二進位（`0x5555`、`0xaaaa`）就是對應到兩種 0/1 交錯的字串，而 0/1 交錯的字串不會用到第二種操作，所以也會維持 dp$[i][0]$ 和 dp$[i][1]$ 的相對關係。

另一方面，$x_{i+K} - x_i$ 看起來就相對沒有明顯的規律，不過這部分可以先放著，因為光有 a_i, b_i 的這個性質就有一些優化空間了。

優化四

這個觀察有什麼用呢？第一個想法可能是想到因為這部分不用打表，K 就可以取得更大，不過不幸的是 x_i 的轉移還是需要查表，而且查表也會需要用到 a_i, b_i，所以還是得算出中間的 a_i, b_i。然而，這個性質卻可以用來消除指令的 dependency：原本每次轉移都需要用到上一次轉移的結果，現在可以例如一次算好 $a_{i+K}, b_{i+K}, a_{i+2K}, b_{i+2K}, \cdots, a_{i+MK}, b_{i+MK}$ 然後把再把這些一次丟去查表。如此消除指令的 dependency 可以有效的利用 CPU 的 pipeline，也因此雖然操作數並沒有變少，執行的時間卻會有很明顯的改進，實際上大約可以拿到 360 分。

支線優化：SIMD

既然在前面都用上 AVX2 的指令了，那不妨就把 AVX2 用到底吧！具體來說可以把跟 C_1, C_2 的比較改成例如 `_mm_max_epu32` 和 `_mm_cmpeq_epi32` 的組合（不能直接用 `_mm_cmpgt_epi32`，因為需要的是無號比較）且查表改成用 `_mm256_i32gather_epi32` 查，還有一些細項的運算也可以用各種 AVX/AVX2 的運算來做。這樣大概可以進步到 380 分左右，不過這個優化和後面的觀察並沒有關係，後面的優化也用不太到這些技巧就是了。

另一個觀察

再來回顧一下在優化四的地方一次計算 $a_{i+K}, b_{i+K}, a_{i+2K}, b_{i+2K}, \cdots, a_{i+MK}, b_{i+MK}$ 的時候，具體是做了什麼樣的操作。例如假設 $K = 8$、S 的長度是 16 bit，計算 b_{i+8} 時是判斷 S 的最低 8 個 bit 是不是小於 `0x55`（或 `0x56`，取決於 b_i）、計算 a_{i+16} 時，是判斷 S 是不是小於 `0x5555`。這個過程看起來有些多餘：在判斷 S 是不是小於 `0x5555` 的過程中，理論上一定會先判斷到 S 的最低 8 個 bit 是不是小於 `0x55`？

如果你學過一點數位電路、知道加法器的概念的話，你可能已經想到要怎麼取得這個中間結果了。想要知道 S 的最低 8 個 bit 是不是小於 `0x55`，其實就是判斷在計算 S 減掉 `0x5555` 的過程中，有沒有從 bit 8 借位；等價地說，是在判斷 S 加上 `0xaaab`（二補數）的時候有沒有進位到 bit 8。而每個 bit 在加法時有沒有被進位，就是看兩個相加的數取 XOR 起來的那個 bit 是否等於結果對應的 bit。也就是說只要把加數、被加數和結果三個數 XOR 起來，就可以得到每個 bit 有沒有進位了！

如此如果用 64-bit 整數的 S（兩個 movemask 的結果接起來），就可以一次算出 $a_{i+2}, b_{i+2}, a_{i+4}, b_{i+4}, \cdots, a_{i+62}, b_{i+62}$。其實要得到 a_{i+64}, b_{i+64} 也不難，可以用 GCC 的 `__builtin_uaddl_overflow` 檢查是否有 overflow（也就是最高位的進位），而算出來的 a_{i+64}, b_{i+64} 可以直接拿來當作下一次迴圈的 a_i, b_i。

最終優化

利用這個觀察，終於可以解決 $x_{i+K} - x_i$ 還是需要查表的問題了！想法是既然現在可以用一次減法就算出所有的 a_{i+2j}, b_{i+2j}，不如就直接取最小值 $K = 2$，因為 $x_{i+2} - x_i$ 只會有 0/1 兩種可能性，就能用一些位元運算直接算出每個 $x_{i+2j+2} - x_{i+2j}$（例如用 `v&v>>1` 就可以判斷 `v` 每兩個 bit 是不是 `11`，用這個方法列出所有 case 就可以了），然後利用 `__builtin_popcount` 全部加起來就可以得到 $x_{i+64} - x_i$ 了！在列出 $x_{i+2j+2} - x_{i+2j}$ 的式子之後可以自己作一些布林運算化簡（不知道為什麼 compiler 好像不會做這件事），可以稍微加速一些。

這就是這題最終的完全體，分數達到了 440 分。如果嘗試寫個只把輸入讀進來完全不做任何額外計算的 code，會發現竟然沒有跟這個解差多少，也就是這個作法幾乎已經是壓常數的極限了！

官解：`https://pastebin.com/61GjtAxF`

pB. 丟硬幣遊戲

這題其實是個出爛的題目。一方面，賽後有人（yenck）想到了一個複雜度比原本解的複雜度還要更好的解[1]，所以這題就失去它壓常數的意義了，然後還花了我一大堆時間才證出這個做法的正確性 Orz 而另一方面，我也沒有意料到會那麼少人寫，我原本的期待是大家會試著 random 模擬唬爛一下然後就發現過了，不過看起來這題 random 的方法並沒有那麼顯然 (?

這題原題是 TIOJ 2076，要計算的東西完全相同，不過這題題敘就是直接把原題的作法講出來了，只留下實作的部分，而且誤差範圍還開大了一些（原題是變異數的相對誤差不超過一成），所以應該明顯簡單很多。底下的題解還是以原本我想到的解為主，最後再給複雜度更好的解。

然後也先在這裡感謝出原題的 hansonyu123 和幫忙提供證明方法，還有 yenck 找到複雜度更好的解。

100 分

這題最關鍵的點在於看清楚題目的誤差範圍：$KL(P \parallel Q) \leq 10^{-2}$ 其實是一個蠻鬆的誤差範圍，就如範例測資的平均差了 10%、標準差差了 5% 還是能夠 AC。既然如此，如果直接模擬這個丟硬幣過程例如 $K = 1000$ 次，然後直接把這 K 次模擬結果的平均和樣本標準差當作答案，應該就有蠻大的機率可以 AC。

不過在模擬之前，具體來說要怎麼實作模擬呢？因為題目的記憶體限制裝不下全部輸入，也當然裝不下全部 M 個硬幣的結果，所以必須要想辦法找到一個方法可以讓我們邊讀輸入邊同時模擬 K 次的丟硬幣結果。可以先想想如果只模擬一次的話需要的東西：總之我們必須有一個方法可以在用很小的空間的情況下，給定一個硬幣編號、回傳該硬幣編號的結果。換個說法，我們需要一個函數 f 使得 $f(i) \in \{0, 1\}$ 是第 i 個硬幣的結果，且 $f(M)$ 的表現要近似隨機、不能用太多記憶體。咦，這不就是 hash function 的定義嗎？於是我們可以找一個品質足夠好的 hash function 當作產生硬幣結果的函數，例如 CF 這篇文章裡面的 `splitmix64`，或者 TIOJ 2059 裡面的 `PRNG` 都是不錯的選擇。

至於要同時模擬 K 次的結果也很簡單：我們只需要 K 個不同的 hash function，而這可以簡單的通過對 hash function 輸入進行簡單的變換達成，例如 $f(i), f(10^9 + i), f(2 \times 10^9 + i), \cdots$ 就可以視為一堆不同的 hash function。（用 $f(i), f(i+1), f(i+2), \cdots$ 是有可能會爛掉的，可以想想看為什麼？）另外一般的 hash function 的輸出通常也不只有一個 bit，每個 bit 也都可以當成各自的 hash function。於是這樣我們只要開個大小為 K 的陣列記錄每次模擬到目前為止的 R，就可以在 $O(NK)$ 時間、$O(K)$ 空間完成模擬，就足

[1] 其實他的解有個小地方沒有分析到，具體來說是差了一個 $\frac{K}{K-1}$ 的 factor，不過影響答案的幅度很小。

夠 AC 了。

其實在算平均和標準差時還可以更聰明一點。題敘中其實有一個小提示：在 Output Format 那邊給出了在 $\mu = \mu_1$ 情況的誤差。事實上，簡單推導可以發現期望值一定會是 $\frac{N}{2}$，因為每個硬幣貢獻的 R 值期望值一定是它出現在序列中次數的一半（要嘛正面要嘛反面），而每個硬幣的結果又是獨立的。在期望值已知的情況下，計算標準差就可以直接用已知的期望值算母體標準差。套用這個技巧可以稍微降低所需要的 K。

更進一步地，我們也可以發現實際上 R 的分布就是一堆獨立的 $p = 0.5$ Bernoulli distribution 的加權加總，套用標準差相加的公式可以得到正確的標準差就是 $\frac{1}{2}\sqrt{\sum_{i=1}^{C} c_i^2}$，其中 c_i 是有多少個 $a_j = i$。不過因為要算 c_i 也需要 $O(N)$ 空間，所以沒有辦法用這個方式算。

事實上，如果用個 64-bit 的 hash function，並把 64 個 bit 都拿來用，然後嘗試找到最低可以 AC 的 K 值（也可以換個 seed 碰個運氣之類的），最後再加個 `#pragma GCC target("avx2")`、`#pragma GCC optimize("O3,unroll-loops")`，大概就能拿到 125 分了。

如果要具體分析 K 要取多少的話，就會需要詳細推導答案的標準差是多少。可以證明相對誤差會是 $O\left(\frac{1}{\sqrt{K}}\right)$，所以 K 大概取到 400 以上就可以過了。具體的證明可以繼續往下看到複雜度更好的解那邊，因為這個解實際上是複雜度更好的解的特例。

優化一

做完上述的幾件優化之後，會發現瓶頸在每讀一個數字就要 $O(K)$ 存取並累加整個計數的陣列。既然如此，或許可以開兩個陣列，不過一個是 8-bit 整數、一個是 32-bit 整數。接著寫個內迴圈每次讀 255 個值，在內迴圈做加總的時候用 8-bit 整數的陣列，跑完 255 個值再把 8-bit 的陣列累加到 32-bit 整數的陣列，最後把 8-bit 的清零，以避免 8-bit 整數不會 overflow。這樣對陣列的累加就會從原本的 32-bit 運算變成 8-bit 會快一些，locality 也會變好。

優化二

接下來有一個簡單的性質：在每個 byte 各自都不 overflow 的情況下，對每個 byte 都做一次加法跟例如把 8 個 byte 視為一個 64-bit 整數做加法的結果會是一樣的。因此這樣就可以把八次 8-bit 加法壓成一次 64-bit 加法。

不過在前一個優化的時候，取決於實作方式，編譯器可能就會套用 SIMD 指令，所以這個優化本身不一定會有明顯的效能改進。但是如果有觀察到這個性質，那我們可以用更快的方法取出 hash 結果：原本是一個一個 bit 讀，現在可以變成把它 AND

`0x010101010101010101` 取出其中八個 bit 然後就可以直接相加了，剩下的 bit 也可以通過把同一個 hash 值右移不同的量再 AND 同樣的值讀取。

同時改用 64-bit 加法了還有額外的好處：這樣就不必受限於 byte 邊界了，可以變成例如以 4 個 bit 當作一個單位，內迴圈改成跑 15 次。甚至再多開一個陣列、加個一層迴圈，中間迴圈跑 17 次，又可以再加速一些。（有沒有在疊 L1、L2 cache 的感覺了？）這個做法大概可以拿到 250 分。

SIMD

目前這些的操作除了 hash 本身以外都是很平行化的，因此可以全部轉換成 SIMD 版本。實作上沒有什麼特別的地方，就是把位移、AND、累加等全部轉換成對應的 AVX2 函數就好了，可以參考 Intel intrinsics 網站尋找需要的函數。這也是原本的官解，分數大概是 320 分。

官解：`https://pastebin.com/JXuJrkje`

複雜度更好的解

接著要來揭曉複雜度更好的解了。這個想法從前面提到的正確答案是 $\frac{1}{2}\sqrt{\sum_{i=1}^{C} c_i^2}$ 的角度會比較好理解。具體來說既然我們沒辦法開出大小為 M 的陣列計算這個平方和，那說不定可以退而求其次，開個大小為 K 的陣列，然後把所有值 hash 到 $[0, K)$，改成計算每個 hash 結果數量的平方和 A，最後再扣掉預期上不同值被分到同一個 bucket 多貢獻的答案。事實上，我們可以證明如果 hash 是隨機的，那麼最後 hash 結果數量的平方和的期望值會是 $\mathbb{E}[A] = \left(1 - \frac{1}{K}\right)\left(\sum_{i=1}^{C} c_i^2\right) + \frac{N^2}{K}$，因此可以透過計算 $\frac{1}{2}\sqrt{\frac{K}{K-1}\left(A - \frac{N^2}{K}\right)}$ 得到最終想要的結果。並且也可以證明這個解法的相對誤差也同樣會是 $O\left(\frac{1}{\sqrt{K}}\right)$ 量級，這樣我們就有一個時間 $O(N)$、空間 $O(K)$ 的解了！證明這些 bound 的詳細推導放在附錄。

另外可以觀察到原本的作法是這個做法的特例：假設取 $K = 2$，可以發現這裡所求的值其實跟模擬的過程計算的標準差的式子一模一樣，只是因為 $K = 2$ 所以相對誤差就是 $O(1)$，需要透過重複 K 次將相對誤差降低到 $O\left(\frac{1}{\sqrt{K}}\right)$。

當然這個解同樣也用得上 SIMD，不過就是變成用來加速 hash 運算。而且這個複雜度已經到分數公式的飽和區了，所以可能只會多得到 0.1 分之類的。

$O(N)$ 解：`https://pastebin.com/hHer5rQj`
SIMD 版本：`https://pastebin.com/kqrMFZd6`

pC. 簡易序列操作

100 分

這題就是個經典的線段樹題，應該沒什麼好解說的吧 (?) 唯一需要注意的小地方是 sum 的模數是 2^{31} 而非 2^{32}，不過線段樹中還是可以全部用 `uint32_t` 做運算讓它自然 overflow、最後回傳答案再取模就好了。注意最好不要用有號整數，因為有號整數 overflow 是 undefined behavior，雖然實務上通常不會有什麼問題。

優化一

線段樹有兩個常數大的點：locality 不太好、遞迴。（順帶一提，遞迴之所以慢的原因是因為會有難以預測的 indirect branch，還有跨遞迴的編譯器優化很難做。）Locality 不好基本上是所有樹型結構的通病，所以或許可以先從遞迴這點下手。

事實上，無遞迴的線段樹也是存在的，就是所謂的 zkw 線段樹。只要實作基本的 zkw 線段樹，把 O3、`unroll-loops` 等開起來，大概就會有 160 分左右的分數。

zkw 線段樹本身其實也有不少可以壓常數的空間。例如 pull/push 迴圈中常見的實作方法是用目前當層的 index 算出下一層的 index，不過這樣會造成指令的 dependency，因此可以改成直接用位元運算（以及 `__builtin_ctz`、`__builtin_clz`）用層數和原本的 index 直接算出 node index，套用上 loop unrolling 就會有明顯的加速。另外為了稍微改進 locality，與其對四個值各開一個陣列，可以把懶標的兩個值綁成一個 struct、另外兩個只有 query 用到的值綁成一個 struct 各開一個陣列，或者四個值全部綁進同一個 struct（兩種各有優缺點，實測效果沒有差很多，不過前者的做法有利於後面的優化）。還有對左右界的 pull/push 可以不用兩個都從 root 開始，其中一邊可以從 LCA 開始就好，而 LCA 也可以直接用位元運算 $O(1)$ 算出來；查詢的時候與其每次都檢查左右界是否交會，也可以改成直接 loop 到 LCA 為止就好。如果加上這些優化，用 zkw 線段樹大概就可以拿到 210 分左右的分數。

到目前為止優化的 code：`https://pastebin.com/2ADP2iJY`

優化二

接下來我們回到 locality 的問題：之所以 locality 不好，是因為在 access 樹上比較接近 leaf 的位置時每次戳的位置都在序列的不同位置，因此可以視為某種程度的 random access。那我們有沒有辦法減少 random access 的次數呢？一個最簡單的方式就是改用 K 元樹，這樣就是減少了 log 的常數，例如如果改用 4 元樹的話 random access 的次數就會

減為一半。

當然改用 K 元樹的話，雖然 locality 會變好，但計算成本自然也會指數上升。要能比二元 zkw 線段樹得到更好的結果，基本上新的 K 元樹必須要能繼承上述提到的所有的優化才能有一點機會。改用 K 元樹還要多算一個東西，就是對於某個 node 一個的左或右界會切到哪個 child 區間。不過可以發現如果 K 取 2 的冪次的話，這些所有的操作都還是能在 $O(1)$ 用位元運算完成。這會需要花一些推導出每種值的公式。把 root node 放在 index 1 可能會稍微好推導一些，不過只是 off-by-one 所以最終的結果和效率都不太有差。

另外實作上為了讓 locality 達到最好，會希望能讓一個 pull 的子操作（例如 pull sum）所存取的記憶體是連續的。如果用一般線段樹的寫法，這會強制把四個值分開放在四個陣列。不過還有一個更好的寫法（更方便後面的優化）：把所有的值往上提一層，變成一個 node 裡面變成存四個長度為 K 的陣列，而 root 的值就省略。這個寫法還有一個附帶的好處就是可以用 `alignas` 確保 node 有 align 到 cache boundary。

不過如果真的實作了 K 元樹，會發現一件事情：變慢了！很明顯運算的提升抵不過 locality 的改善。不過幸好 K 元樹還有另一個好處，就是我們可以用得上已經在這份題解出現第 N 次的東西：SIMD。

SIMD, Again

其實如果對 SIMD 有一點概念，在想到多元樹的時候應該就會很自然地想到 SIMD，畢竟現在每個 node 都是有很多可以平行處理的元素。實作上就是把 push、pull 和算答案各自用對應的 SIMD 操作取代即可，當然免不了就又會讓 code 充滿各種神秘的函數。有幾個實作上會需要注意的地方：

1. 實作的時候會需要單一元素存取、也會需要整個 SIMD 的操作。為了不要一直做 type conversion，可以用一個 `union` 把一個陣列和一個 `__m256i` 放在一起。

2. 在線段樹上會需要用到各種修改某個連續區間的子樹，或者取得某個子區間的答案。這可以透過 `_mm256_blend_epi32` 達成。Blend mask 的計算會比較複雜一些，可以用 `constexpr` 打好每個左右界對應的 mask 的表就可以直接用了。

3. 因為要把一個 SIMD register 裡面的每個元素做 reduce（例如把其中的 8 個 32-bit 值加起來）是比較費時的操作，因此在 query 的時候，應該是把每個 node 的答案用 SIMD 操作 merge 起來，最後再做一次 reduce 操作，而不是每個 node 都先 reduce 再 merge。

4. Reduce 操作不一定要自己用 intrinsics 寫，直接用單純的迴圈寫 compiler 通常也會把它直接優化成 SIMD 版本。不過有時候神秘的 compiler 優化在某些神秘的條件下不

會做這個優化，所以最好看一下編譯結果確認。

5. 寫的時候最好寫個通用不同 K 的版本，這樣可以實驗看看 K 取多少會最好。

自己的實作結果是 $K = 16$ 最快，可以拿到 370 分左右，不過 $K = 8$、$K = 32$ 也都可以拿到 360 分。

官解：`https://pastebin.com/sAhUBUEx`

pD. 巨集計算

這題是五題中最複雜、但也分數最高的一題。

100 分

就照著題敘把 11 種操作寫出來就可以了，是因為太簡單還是題敘太長所以沒有人做嗎 (?

小優化

一個小優化是用 `switch` 會比用 `if` 還快。這是因為兩者產生出來的 code 不同：`if` 會產生連續的判斷，而 `switch` 會產生 jump table，也就是把每種操作的 code address 存到一個表格裡面，執行到的時候就直接查表跳到該 address 執行。因為這題的 case 足夠多，所以後者會稍微快一些。另外，如果再額外對常數和 memory 版本分 case（例如對 $2X + p$ 做 switch），因為少了一個判斷 p 的 branch，也會稍微變快一些。

巨集優化

接下來的優化就是要就題目敘述所說，從題目所給的範例巨集來做優化了。這部分是個非常純粹的 case-by-case 優化，也是想說模仿實際應用的時候可能會需要針對特定 data 做優化的情況。

Dependency Graph

雖說是 case-by-case，但是還是有一些通用的優化可以先做。例如在 01、02、04 幾個巨集當中，可以發現都有一些多餘的操作，只是形式不太一樣：01 是有一些記憶區的值根本不會被用到、只會被寫入，所以除了最後幾次執行巨集之外都可以省略；02 是有

不少記憶區被寫入之後馬上就被覆蓋掉；04 則是有不少運算是對 m0、m1 兩個常數進行運算，這些值理論上可以事先算好。這幾個 case 都可以用同樣的方法處理。

可以考慮構造這些運算的 depedency graph：把每一次運算的結果和一開始的 8 個記憶區都建立一個點，然後建出從該運算用到的值對應的點到它的有向邊。而因為有迴圈的關係，最後再把執行到巨集最後 8 個記憶區對應的點建一條邊連回一開始 8 個記憶區的點。可以注意到這個圖中，如果有一些點不能走到任意一個 SCC，代表它不 depend on 中間的運算過程，有可能是 01 的狀況，或者是 02 那種會被覆蓋掉的運算；如果有一些點不能被任何 SCC 走到，那麼它除了最開始的幾次 iteration 可能會改變，其餘就會保持常數的狀態，因此可以被事先計算出來。

因此，對於這幾種 case，可以構造這個 dependency graph，找出其中的 SCC，然後把除了不能被 SCC 走到或不能走到 SCC 的節點以外所有的運算弄成另外一個化簡版的巨集。有些保留下來的節點會用到被刪掉的節點對應的常數，這些節點就要變換成適當操作的常數版本（可能還會需要新增幾種額外的操作，例如常數減掉記憶區的值就無法用題目中的巨集形式表示）。做完這件事後，之後每次 `run_macro` 時，首先先執行個 8 次原本的巨集讓所有的常數固定下來，把這些常數放進化簡版的巨集裡面，執行 $T-16$ 次化簡版的巨集，最後再執行 8 次原本的巨集算出只有被寫入的那些記憶區的值。

Barrett reduction

在 03 這個 macro 中，我們看到了大量的除以常數或模常數的運算。由於在電路層面很難平行化，除法是一個非常緩慢的指令，執行時間通常會是其他指令的好幾倍。不過在大量除以同一個常數的情況下，我們可以先求出這個常數的「倒數」，然後改用乘法取代除法。

具體來說，在除以或模常數的情況下，根據除數有兩個很明顯的 case 可以先特判掉：第一是被除數為 0，可以直接轉換成 assign 0；第二是被除數為二的冪次，可以直接轉換為位元右移或 AND 操作。對於其他 case，會希望把除數 x 預處理成一個 “magic number” $m=\left\lceil\frac{2^k}{x}\right\rceil$，使得可以藉由計算 $\left\lfloor\frac{am}{2^k}\right\rfloor$ 得到 $\left\lfloor\frac{a}{x}\right\rfloor$ 的值，如果是取模再由 $a-x\left\lfloor\frac{a}{x}\right\rfloor$ 得到結果。詳細推導出 k 的過程這裡省略，不過最後的結果會得到兩種 case（以下以 highbit(x) 代表最高位的位置，等價於 `31-__builtin_clz(x)`）：

1. $k=33+\text{highbit}(x)$：這個 case 可以適用於所有除數 x，不過會稍微花多一點時間：因為 m 會有 33 個 bit，會導致 am 沒辦法用 64-bit 乘法計算，因此需要改成儲存 $m'=m-2^{32}$，然後最後用 $\left\lfloor\frac{\lfloor m'a/2^{32}\rfloor+a}{2^{k-32}}\right\rfloor$ 計算結果。中間的加法結果也會有 33 個 bit，所以要用 64-bit 的運算。（有另外一種算法是 $\left\lfloor\frac{\lfloor(a-m')/2\rfloor+m'}{2^{k-1}}\right\rfloor$ 就可以全部都用 32-bit 運算，不過在這個 32-bit 的 case 會稍微慢一些，所以這個式子通常被用在 64-bit 的除法。）

2. $k = 32 + \text{highbit}(x)$：這個 case 只適用於部分的除數，具體的條件是 $(2^{32}-1)(mx-2^k) < 2^k$。不過因為 m 剛好是 32 個 bit，所以可以直接用 64-bit 乘法計算 $\left\lfloor \frac{am}{2^k} \right\rfloor$。

注意如果要分這兩個 case 的話，必須要把這兩個 case 也整合進 switch statement 裡面，不然在 case 裡面多一個額外的 branch 判斷，有可能反而會變得更慢。

線性遞迴

05、06 這兩個 macro 看起來十分特別：一個只有加減法和乘以常數，一個只有 XOR 和位元左右移。而這樣的形式代表每次的巨集都是個線性運算，也因此求重覆執行的結果就是求解線性遞迴。加減法和乘常數就是 8 個記憶區加一個常數的線性遞迴、而 XOR 和位元左右移則是 256 個 bit 加一個常數的線性遞迴。因此，這就可以使用矩陣快速冪在 $O(\text{矩陣乘法} \times \log T)$ 的時間內得到最後的值。XOR 的矩陣乘法因為是 0/1 矩陣，還可以用 bitset 加速運算。

這個優化可以跟前面 dependency graph 的分析結合：其實不需要整個巨集所有的運算都是符合條件才能用，只要在拿掉多餘的 node 之後 dependency graph 的其中一個弱連通分量（把有向邊變無向邊後的連通塊）中所有運算都符合條件，代表它計算過程不會用到別的 SCC 的值、也沒有 SCC 會用到它的值，就可以針對這個弱連通分量做線性遞迴的優化了。

另外，XOR 的線性遞迴刻意留了個小 caveat 沒有寫在巨集裡面：就算再加上位元取反和 AND 常數兩種操作，也還是符合 XOR 的線性遞迴。

Cycle finding

07 這個 macro 最明顯的特徵是只用了兩個記憶區：事實上不只如此，連 `memory[1]` 的值都會被直接覆寫掉，所以最終結果只會被一個記憶區的值影響。既然如此，這個 macro 的運算看起來也很 random，或許可以預期大概 2^{16} 個 iteration 就可能會進入循環了？於是我們可以對這種只有用到一個記憶區的 case 實作個 $O(1)$ 空間的 cycle finding algorithm。

不過我們並沒有辦法保證只用一個記憶區就一定會有短的循環節，事實上也很容易構造會把 2^{32} 個值全部跑過一次的 cycle。因此最好的做法是邊計算邊找循環節，而 Brent's algorithm 就非常適合，只要在每個 iteration 跑完之後多一個 if 判斷就可以了！這樣就算沒有找到循環節，實際執行的效率也不會有什麼差異。

這個優化也同樣可以搭配 dependency graph 的分析，就是看在移除多餘 node 之後有沒有一個孤立的 SCC 裡面只包含一個記憶區。

小結

如果你把以上四個東西全部實作出來，那麼大概能夠得到 300 分左右。不過實作的複雜度不是一般的高，所有 case 都寫完的話大概要寫 600~700 行的 code。

把這些寫完的 code：`https://pastebin.com/4rxaP2nh`

編譯

如果你有嘗試把題目給的 7 種 macro 直接 hardcode 在 code 裡面，你會發現就可以在對應的 macro 拿到分數。不過明明是做同樣的事情，為什麼模擬和 hardcode 的效率差這麼多？除了編譯器優化可能會把多餘的操作優化掉以外，更重要的是模擬的問題：模擬的過程中每次執行一個操作都要先判斷是哪一種 operation，而這個 switch 就是在執行一個 indirect jump，這種 jump 因為沒辦法做 branch prediction，所以常數很大。

既然如此，那有沒有可能在拿到巨集的時候預處理好某些東西，讓之後執行巨集時就不需要每次判斷操作種類了？答案是可以的，而做法就是寫一個簡易的編譯器！具體來說，在 `init_macro` 的時候把給定的操作編譯成 machine code，把它放在一個陣列裡面，用 `mprotect` system call 把這個陣列的空間標示成可以執行，接著用一個函數指標指向這個陣列，這樣只要產生出來的 code 符合 calling convention 就可以當作正常的函數使用了。

要自己實作編譯器，免不了就要進入組合語言的黑魔法境界了 (?) 為了最高的效率，會希望把 8 個記憶區全部放在 register 裡面直接運算，而 x86-64 總共有 16 個 register，扣除掉 eax、edx 會用來執行除法、ecx 會用來執行左右移，還有一個用來存放 loop 次數的 register，以及運算過程為了移動記憶區的值也會需要一個額外的 register，剩下的 register 仍然十分夠用。不過因為有 callee saved register 的關係，為了能使用這些 register，function 一開始要先把 callee saved register 先 push stack，最後再 pop 回來。

於是要產生的 function 大概可以分為三個部分：一開始（prologue）push stack、把 memory 陣列載入到 register 中，另外可以再加個 no-op padding 讓 loop align 在 cache line boundary；第二部分就是主要的 loop，就是產生巨集的各種 operation，最後 increment loop variable 檢查是否要繼續迴圈；最後一個部分（epilogue）把 register 重新寫回 memory、把 stack pop 回來並且 return。

作為一個以 CISC 著稱的 instruction set，x86-64 的 instruction encoding 也不是一般的複雜，要寫出這個小編譯器也需要花不少時間研究要怎樣才能產生所需要的指令。不過所幸的是這題的操作都算簡單，只有除法和取模要特判 0 稍微複雜一些，而且都固定只用 32-bit register，所以只會用到少數幾種 encoding。而 function prologue、epilogue 這兩段固定的東西都可以例如先用 `as` 和 `objdump` 產生好並 hard code 在程式裡面，這樣就不需要管一些 encoding 比較複雜的 memory load/save instruction。

總之如果你成功寫出這個 compiler 的話，就算前面那些運算化簡、針對性的優化全部不做，也大概可以拿到 350 分左右。

順帶一提，這件事情某種意義上也可以視為一種 virtual machine，在執行把這種巨集轉換成 x86-64 的 static binary translation。

純編譯沒有前面優化的 code：`https://pastebin.com/0ePYJkwh`

最終優化

最終優化其實就是把針對各巨集特性的優化和 compiler 結合在一起就好了，這樣大概可以拿到 600 多分。給到這麼多分自然是因為要實作的東西劇多（SCC、DFS、disjoint set、constant folding、cycle finding、linear & xor recurrence 還有一個 x86-64 assembler），官解的 600 分解整個寫下來的 code 達到了驚人的 930 行 (!)，也是這幾題中完全體最肥的一題。

寫到這裡，如果把前面的建圖當成 intermediate representation 的話，其實這已經可以視為是一個 middle-end 到 backend 的編譯器了（其實建 SCC 的過程也有點像編譯器中會用到的 SSA 形式，只不過是超級簡化版）。這題也因此還有各種可以用編譯器技巧優化的空間，例如用 lea 或帶 immediate 的 instruction 減少 instruction 數量，或者跑個 register coalescing 減少 mov 使用量、跑個 instruction scheduling 增加 pipeline 效率之類的，有興趣了解一些編譯器原理的可以嘗試看看 (?

930 行的官解：`https://pastebin.com/uHUN12Lk`

pE. 旅遊

這題是唯一給分比較不太一樣的題目。因為這題原題本身就需要小壓一點點常數，所以給分上抓成如果實作到每筆測資都不 TLE 的話就會拿到至少 150 分左右。

另外這題是自己在 NPSC 2020 出的題目，但是當年賽後題解的時候我給的複雜度分析是爛的，特此道歉 (?) 當年爛掉的複雜度分析放在附錄，有興趣的可以嘗試在看底下的題解之前先看一下這個爛掉的複雜度分析，看你能不能找出爛在哪裡 (?

原題

這題有兩種暴力解：一種是對所有 degree d 找 (a_i, a_{i-1}) 在 mod d 底下的循環節，複雜度是所有 degree 的平方和；另一種是對每個 degree 都 $O(T)$ 暴力算，因為 degree 最多會有 $\sqrt{E}$ 種，複雜度是 $O(T\sqrt{E})$。

而這兩種解法合併就是正解：會希望對於每一種 degree，各自挑會花比較少時間的方法來做，因此對於 degree 超過 $K = O(\sqrt{T})$ 的用暴力做會比較好，其它的找循環節會比較好。至於時間複雜度會需要仔細分析一下。對於找循環節的部分，如果 T 的量級比 E 大，那麼最多會有 $O\left(\frac{E}{\sqrt{T}}\right)$ 種 degree，否則只會有 $O(\sqrt{T})$ 種 degree，每個要花 $O(T)$ 時間，因此這部分的複雜度是 $O(\min(E,T)\sqrt{T})$。對於暴力的部分，如果 T 的量級比 E 大，則最多會有 $O\left(\frac{E}{\sqrt{T}}\right)$ 種 degree，否則最多會有 $O(\sqrt{E})$ 種 degree，每個一樣花 $O(T)$ 時間，因此這部分的複雜度是 $O(\min(E\sqrt{T}, T\sqrt{E}))$。兩個加總，得到最終複雜度是 $O(\min(E\sqrt{T}, T\sqrt{E}))$。

這題本身就有一些基本的常數要壓，例如找循環節要用二維陣列，不能用 `unordered_map` 或者 $O(1)$ 空間演算法（因為會造成多算很多次遞迴式），以及計算那個遞迴式的時候要把整個式子算出來最後再取模。這兩者做到就可以拿到 AC 150 左右。微調一下算循環節和暴力算的分界可以拿到稍微多一點分數。

優化一

這題的瓶頸就是計算那一串遞迴式本身，而其中最慢的操作又屬其中的取模。又由於取模的數字是固定的，我們可以套用前一題用到的 Barrett reduction。不過有幾個差異：

1. 這裡是 64-bit 整數的除法，所以所有的整數大小都要加倍，例如乘法就要變成 128-bit 的，因此會用到 `__int128`。

2. 因為這題不是要編譯，要採用哪種 case 會變成在 runtime 決定，如果還去分 case 的話會造成額外的 branch，所以實作通用的 case 就好了。

3. 在 64-bit 上實作通用的 case 就會用到那個比較複雜的算法（因為 128-bit 的加法會產生更多 instruction）。不過這裡可以利用那個遞迴式的結果其實不會超過 63 bit 的特性，這樣 $\left\lfloor \frac{m'a}{2^{64}} \right\rfloor + a$ 也就不會超過 64 bit，因此可以保持用比較快的算法並用 64-bit 加法即可。

把所有取模（包含算遞迴式本身，以及算出循環節之後每次要 access 循環節裡面某個位置會用到的模運算）都套用 Barrett reduction 之後，大概可以拿到 210 分。

優化二

另一個簡單的優化是可以把不同的 degree 取它們的 LCM 或直接乘起來當作一個 degree 來算，這樣就有機會減少所需的計算量。不過要併哪些 degree 也是要分析一下的：我們不會希望把兩個 degree 太小的點 a, b 併起來，因為原本算循環節只是

$O(a^2+b^2)$，合併起來就會變成 $O(a^2b^2)$。因此，會希望參與合併的 degree 至少有一個是原本就是暴力算的，或者原本是算循環節但是 degree 非常接近 K。另外，合併的時候也要控制在遞迴式不會超過 63-bit，這樣才能適用上述的 Barrett reduction，否則會變成算遞迴式要多作幾次 mod 而變得更慢。列個二次方程式求解可以得到條件是模數不要超過 5544778。

一個簡單的 heuristic 是從最大的 degree 開始掃，然後找到合併之後不會超過模數且 degree 最大的一個點跟它合併，如果沒有就再試次小的 degree，一直試到 K 或比 K 小一點的 degree 為止。這個解法可以拿到 240∼250 分。

小觀察

接下來有一個神秘的觀察，不過這個觀察我沒有證明、也不確定是不是真的完全成立，而且可能要真的嘗試去生個測資才會發現（或者你有參加過 NPSC 2020 而且還記得當年的題解）。題目中的遞迴二次式看起來十分的隨機，而可以想像如果對於一個 N^2 個點的圖每個點都隨機指向另一個點，隨機挑一個起始點開始走，期望上應該走 $O(N)$ 步就會進入循環（因為不進入循環代表每一步都不能走到之前走過的點，類似 birthday paradox），那麼理論上要構造出一個初始條件和 A, B, C, D, E, F 使得對於每個 degree 循環節都達到 $O(d^2)$ 應該是一件非常困難甚至不可能的事情（或許能有一兩個 degree 可以達到 d^2，但要同時讓所有 degree 同時達到就非常困難）?

事實上這個猜想就自己觀察幾乎是對的，只有一個例外：$B=C=0$ 的情況下幾乎所有 random 產生的測資都能讓每個點的循環節長度達到 d^2 量級。不過經過了這麼多年我還是沒有找到證明，或至少一個直覺的 heuristic 能夠大概解釋這為什麼會成立，有興趣的人可以思考看看。一個可能可行的想法是嘗試證明在 $B=C=0$ 的情況下多個 node 指向同一個 node 的比例會顯著比 random 小，而其他的條件則沒有這個性質。如果有人對此有想法的話歡迎跟我講 (?

雖然沒有證明，但還是不妨礙把這個性質拿來優化。具體來說，如果發現 $B=C=0$，那算遞迴式就可以省掉兩次乘法，然後還會因為遞迴式取模的最大值變小而變成可以合併更大的 degree。而如果 $B\neq 0$ 或 $C\neq 0$，則因為預期多數循環節的長度很短，就可以把找循環節的分界調高。這個做法可以拿到將近 260 分。

這題或許還有一些可能的優化空間，例如可以找一些 gcd 比較大的 pair 取它們的 LCM 合併起來，這樣或許可以多合併一些數字，不過 heuristic 比較不確定要怎麼設計比較好，所以有點懶得嘗試 (?

官解：`https://pastebin.com/DHbG0ghm`

附錄一：pB 證明

令隨機變數 $x_{ik} \in \{0,1\}$ 代表 i 這個數字經過 hash 之後是不是等於 k（因此會符合 $\sum_k x_{ik} = 1$），並令 c_i 代表 i 這個數字出現幾次，另外用 $|\mathbf{c}|^2$ 表示 $\sum_i c_i^2$。可知 $\sum_i c_i = N$，而 $\frac{\sum_i c_i^2}{4} = \frac{|\mathbf{c}|^2}{4}$ 即是原題所求的變異數。這裡想要求的是被 hash 成同一個數字的數量的平方和 A 的期望值和標準差。

為了版面的整潔，以下的所有 summation，如果有多個同一種變數出現在同一個 $\sum$ 底下（這裡有兩種變數：hash 前的數字和 hash 後的數字，前者會用 i, j、後者會用 k 作變數以便區分）且**用括號括起來**，則代表被同一個括號括起來的變數相等、在不同括號的變數不相等。例如 $\sum_{(i)(j),k_1,k_2}$ 代表 $\sum_{i\neq j,k_1,k_2}$，$\sum_{(i_1,j_1)(i_2)(j_2),k}$ 代表 $\sum_{i_1=j_1,i_1\neq i_2,i_1\neq j_2,i_2\neq j_2,k}$。

先求期望值。首先我們有

$$A = \sum_k \left(\sum_i x_{ik}c_i\right)^2 = \sum_{i,j,k} x_{ik}x_{jk}c_ic_j = \sum_{i,k} x_{ik}^2c_i^2 + \sum_{(i)(j),k} x_{ik}x_{jk}c_ic_j$$

利用 $\mathbb{E}[x_{ik}^2] = \frac{1}{K}$、$\mathbb{E}[x_{ik}x_{jk}] = \frac{1}{K^2}$（$i \neq j$）可得

$$\begin{aligned}
\mathbb{E}[A] &= \sum_{i,k} \mathbb{E}[x_{ik}^2]c_i^2 + \sum_{(i)(j),k} \mathbb{E}[x_{ik}x_{jk}]c_ic_j \\
&= \sum_{i,k} \frac{c_i^2}{K} + \sum_{(i)(j),k} \frac{c_ic_j}{K^2} \\
&= \frac{1}{K}\cdot K\sum_i c_i^2 + \frac{1}{K^2}\cdot K\sum_{(i)(j)} c_ic_j \\
&= |\mathbf{c}|^2 + \frac{1}{K}\left(\left(\sum_i c_i\right)^2 - \sum_i c_i^2\right) \\
&= \left(1 - \frac{1}{K}\right)|\mathbf{c}|^2 + \frac{N^2}{K}
\end{aligned}$$

因此 $\frac{1}{4}\frac{K}{K-1}\left(A - \frac{N^2}{K}\right)$ 的期望值的確就是所求的變異數 $\frac{|\mathbf{c}|^2}{4}$。

接著來求這個估計值的不確定性。為了方便，我們可以先計算 A 的標準差，也就是 $\sqrt{\mathbb{E}[A^2] - \mathbb{E}[A]^2}$。照樣，我們可以得到

$$\mathbb{E}[A^2] = \sum_{i_1,i_2,j_1,j_2,k_1,k_2} \mathbb{E}[x_{i_1k_1}x_{j_1k_1}x_{i_2k_2}x_{j_2k_2}]c_{i_1}c_{i_2}c_{j_1}c_{j_2}$$

接著，我們可以用 i_1, i_2, j_1, j_2 的 equivalence class（也就是哪些之間相等、哪些之間不相等的關係）分類，求出每種情況的 $\mathbb{E}[x_{i_1k_1}x_{j_1k_1}x_{i_2k_2}x_{j_2k_2}]$。以下的表格中的 equivalence class 也使用和 summation 底下相同的格式（同一個括號中的變數為相等，不同括號中的

變數不相等）：

	(i_1,i_2,j_1,j_2)	(i_1,j_1) (i_2,j_2)	(i_1,i_2) (j_1,j_2)	(i_1) (i_2,j_1,j_2)	$(i_1)(j_1)$ (i_2,j_2)	$(i_1)(i_2)$ (j_1,j_2)	$(i_1)(j_1)$ $(i_2)(j_2)$
$(k_1)(k_2)$	0	$1/K^2$	0	0	$1/K^3$	0	$1/K^4$
(k_1,k_2)	$1/K$		$1/K^2$	$1/K^2$		$1/K^3$	
等價的 equiv. class 數量	1	1	2	4	2	4	1

有了這張表，就可以展開上面那一大坨 summation 了：

$$
\begin{aligned}
\mathbb{E}[A^2] = &\sum_{i_1,i_2,j_1,j_2,k_1,k_2} \mathbb{E}[x_{i_1k_1}x_{j_1k_1}x_{i_2k_2}x_{j_2k_2}]c_{i_1}c_{i_2}c_{j_1}c_{j_2} \\
=&\sum_{i,k}\frac{1}{K}c_i^4 + \sum_{(i)(j),k_1,k_2}\frac{1}{K^2}c_i^2c_j^2 + 2\sum_{(i)(j),k}\frac{1}{K^2}c_i^2c_j^2 \\
&+4\sum_{(i)(j),k}\frac{1}{K^2}c_ic_j^3 + 2\sum_{(i_1)(i_2)(j),k_1,k_2}\frac{1}{K^3}c_{i_1}c_{i_2}c_j^2 + 4\sum_{(i_1)(i_2)(j),k}\frac{1}{K^3}c_{i_1}c_{i_2}c_j^2 \\
&+\sum_{(i_1)(i_2)(j_1)(j_2),k_1,k_2}\frac{1}{K^4}c_{i_1}c_{i_2}c_{j_1}c_{j_2}
\end{aligned}
$$

為了避免等一下的式子變得太肥，以下先定義一個 vector

$$
\mathbf{x} = \begin{bmatrix}\sum_i c_i^4 & \sum_{(i)(j)} c_i^2c_j^2 & \sum_{(i)(j)} c_ic_j^3 & \sum_{(i_1)(i_2)(j)} c_{i_1}c_{i_2}c_j^2 & \sum_{(i_1)(i_2)(j_1)(j_2)} c_{i_1}c_{i_2}c_{j_1}c_{j_2}\end{bmatrix}^T
$$

注意我們有

$$
\begin{aligned}
N^4 &= \left(\sum_i c_i\right)^4 = \begin{bmatrix}1 & 3 & 4 & 6 & 1\end{bmatrix}\mathbf{x} \\
N^2|\mathbf{c}|^2 &= \left(\sum_i c_i^2\right)\left(\sum_i c_i\right)^2 = \begin{bmatrix}1 & 1 & 2 & 1 & 0\end{bmatrix}\mathbf{x} \\
|\mathbf{c}|^4 &= \left(\sum_i c_i^2\right)^2 = \begin{bmatrix}1 & 1 & 0 & 0 & 0\end{bmatrix}\mathbf{x}
\end{aligned}
$$

把上面一大串 summation 以此化簡，可得

$$\begin{aligned}
\mathbb{E}[A^2] &= \begin{bmatrix} 1 & 1+\frac{2}{K} & \frac{4}{K} & \frac{2}{K}+\frac{4}{K^2} & \frac{1}{K^2} \end{bmatrix} \mathbf{x} \\
&\le \begin{bmatrix} 1+\frac{2}{K}-\frac{1}{K^2} & 1+\frac{2}{K}+\frac{1}{K^2} & \frac{4}{K} & \frac{2}{K}+\frac{4}{K^2} & \frac{1}{K^2} \end{bmatrix} \mathbf{x} \\
&= \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{x} + 2\left(\frac{1}{K}-\frac{1}{K^2}\right) \begin{bmatrix} 1 & 1 & 2 & 1 & 0 \end{bmatrix} \mathbf{x} + \frac{1}{K^2} \begin{bmatrix} 1 & 3 & 4 & 6 & 1 \end{bmatrix} \mathbf{x} \\
&= |\mathbf{c}|^4 + \frac{2}{K}\left(1-\frac{1}{K}\right) N^2 |\mathbf{c}|^2 + \frac{N^4}{K^2}
\end{aligned}$$

因此

$$\begin{aligned}
\mathbb{E}[A^2] - \mathbb{E}[A]^2 &\le |\mathbf{c}|^4 + \frac{2}{K}\left(1-\frac{1}{K}\right) N^2 |\mathbf{c}|^2 + \frac{N^4}{K^2} - \left(\left(1-\frac{1}{K}\right)|\mathbf{c}|^2 + \frac{N^2}{K}\right)^2 \\
&= \left(1-\left(1-\frac{1}{K}\right)^2\right)|\mathbf{c}|^4 \\
&= \frac{2K-1}{K^2}|\mathbf{c}|^4 \le \frac{2}{K}|\mathbf{c}|^4
\end{aligned}$$

因此 A 的標準差不超過 $\sqrt{\mathbb{E}[A^2]-\mathbb{E}[A]^2} \le \sqrt{\frac{2}{K}}|\mathbf{c}|^2$，也就是有高機率可以以此得到相對誤差 $O\left(\frac{1}{\sqrt{K}}\right)$ 的答案。注意這是求出來變異數的標準差，如果用一階近似的話可以大概估計標準差的標準差約略是變異數的標準差的一半。

附錄二：pE 爛掉的複雜度分析

考慮給定常數 K，對於 degree 超過 K 的用暴力做，其它找循環節。

已知 degree 不超過 K 的 degree 種類數不會超過 $O(\min(K, \sqrt{E}))$ 種，每個循環節長度不超過 K^2，所以這部分複雜度是 $O(\min(K^3, K^2\sqrt{E}))$。

另一部分，degree 超過 K 的點數不會超過 E/K 個，每個要花 $O(T)$ 暴力做，所以這部分複雜度是 $O(TE/K)$。也因此總複雜度是 $O(\min(K^3, K^2\sqrt{E}) + TE/K)$。

我們考慮把兩個 min 的部分分開，各自用算幾不等式找最佳的 K。對於 $O(K^3 + TE/K)$，可以取 $K = O(\sqrt[4]{TE})$，可化簡為 $O((TE)^{3/4})$；對於 $O(K^2\sqrt{E} + TE/K)$，可以取 $K = O(T^{1/3}E^{1/6})$，可化簡為 $O(T^{2/3}E^{5/6})$。也因此，最後的總複雜度就是 $O(\min((TE)^{3/4}, T^{2/3}E^{5/6}))$。