1 數論與快速傅立葉轉換

本章的作者們都具有數學背景，內容與符號使用上會偏數學一點，因此以下將介紹一些數學背景知識。倘若學員往後要更深入微積分和線性代數，一定可以更加上手！

文中使用*定義*（Definition），以一段簡短的敘述規範一個詞彙或一個概念的意義，建立起讀者與筆者間互相溝通的語言。通常在限定討論內容的範疇時，會使用*集合*指定列舉具有某種性質的總體，集合的呈現有時候會用大寫符號，或是用 $\{\}$ 夾住正在討論的內容。例如 $\mathbb{Z}$ 代表所有正負整數與 0，或用 $\{a, b, c, d, e, \ldots, z\}$ 表示所有小寫英文字母。以正式表示法而言，一個元素至多在集合中只能出現一次（不過可以做一些變化讓一個元素出現兩次以上）。在了解集合的定義以後，下文就可以一直使用相同符號代表與定義相同的意義。集合裡面的東西我們稱作*元素*，用 $\forall x \in \mathcal{S}$ 代表對於所有在集合 $\mathcal{S}$ 中的元素，用 $\exists x \in \mathcal{S}$ 代表存在在集合 $\mathcal{S}$ 中的元素是我們要討論的對象。

數學家不喜歡「大概」、「應該」這類的詞彙，當有些性質已經是一定正確（機率 100% 還不保證一定正確喔），我們會用*定理*（Theorem），*引理*（Lemma）來表示一個已經確切知道的現象，通常定理會附上*證明*（Proof）來闡述我們的思路（有時候將證明思路修改之後，就會變成解題的關鍵），不過這裡的證明只會寫上證明思路，不會附上完整證明，而有些定理證明需要的預備工具太多，在此處會略過。

對於一件可以是真是假的事情，我們會說它是一個*性質*，性質與性質之間可以有關係，這個關係也是一個性質。例如，（我不出門）是一個性質，（下雨）也是一個性質，（如果下雨，我就不出門）也是一個性質。性質與性質之間可以有等於或不等於的關係，例如，（如果我出門，那一定沒下雨）等於（如果下雨，我就不會出門）。而定理是一個恆為真的性質，$(A) \Rightarrow (B)$ 代表，如果 (A) 成立，則 (B) 也一定會成立。$(A) \Leftrightarrow (B)$ 代表兩者不是同時成立，就是同時不成立。

本章有些演算法會附上程式碼，其難易度並不取決於程式碼長度。程式碼中的命名都不短，目的是希望讀者可以透過檢視其命名就能了解意義，不需再額外註解。此外講者在撰寫本章時，考慮到整體理論表現的完整度，因此將一些測試用的程式碼搭配在一起，讓讀者可以自行比較兩者的差異。

1.1 數論 Number Theory

競賽中數論問題通常會與一些性質結合，本節會講解一些基本的數論定義、定理，以及利用藉由定理基礎寫出競賽中能夠使用的演算法。

1.1.1 整除性 Divisibility

定義 1.1. *對於兩個整數 a, b，我們說 a **整除** b，或 b 被 a 整除，若存在一個整數 z 使得 $az = b$，記作 $a|b$，此時我們稱 a 是 b 的**因數**，b 是 a 的**倍數**。*

定義 1.2. *對於一個大於 1 的整數，如果其因數只有自己和 1，則稱這個數是**質數**，否則稱為**合數**。在此定義中 1 既不是質數也不是合數。*

定理 1.1. *n 是一個合數。存在一個整數 d 使得 $d|n$ 且 $d \leq \sqrt{n}$*

Proof. n 是一個合數代表可以寫為 $n = ab$。若 $a, b > \sqrt{n}$，則 $ab > n$。明顯矛盾！ □

定理 1.1 提供了一種簡單的質數測試，給定一個大於等於 2 的正整數 n，可以測試所有小於等於 $\lfloor\sqrt{n}\rfloor$ 的整數，是否整除 n。若存在 a 使得 $a|n$，則 n 是合數，否則為質數。時間複雜度為 $\mathcal{O}(\sqrt{n})$。

```
bool isPrime(int n) {
  for (int i = 2; i * i <= n; ++i)
    if (n % i == 0) return false;
  return true;
}
```

程式碼 1.1: 樸素的質數測試

定義 1.3. *對於一個不知道是否是對的性質 $\mathcal{P}$，使用此符號表示：*

$$[\mathcal{P}] = \begin{cases} 1 & \text{性質}\mathcal{P}\text{是正確的} \\ 0 & \text{性質}\mathcal{P}\text{是錯誤的} \end{cases}$$

定義 1.4.

$$\sum_{d|n} f(d) := \sum_{x=1}^{n} [x|n] f(x)$$

有一點值得注意的是

$$\sum_{d|n} H\left(\frac{n}{d}\right) = \sum_{d|n} H(d)$$

1.1.2 埃氏篩法 Sieve of Eratosthenes

考慮以下這種題型：

$$\sum_{n=1}^{N}\sum_{p=1}^{n} f(p,n)\,[p\text{ 是質數}]\,[p|n]$$

限制是 $N \leq 10^5$

廣義而言，我們用 `doSomeThing(p, n)` 作為對一些 p, n 進行處理，注意執行順序不能影響答案。在這個例子中是在總和加進 `f(p,n)`，並假設 `isPrime(p)` 可以在常數時間內判斷 p 是不是質數，此時的程式碼會是：

```
int ans = 0;
void doSomeThing(p, n) {
  ans += f(p, n);
}
for (int n = 1; n <= N; ++n) {
 for (int p = 1; p <= n; ++p) {
   if (isPrime(p) && n % p == 0) {
     doSomeThing(p, n);
   }
 }
}
```

程式碼 1.2: 硬解

複雜度分析：$\mathcal{O}(n^2)$，吃了一個大大的 TLE 饅頭。

注意到此處若將 $\sum$ 交換可得到

$$\sum_{p=1}^{N}\sum_{n=p}^{N} f(p,n)\,[p\text{ 是質數}]\,[p|n] = \sum_{p=1}^{N}\,[p\text{ 是質數}]\left(\sum_{n=p}^{N} f(p,n)[p|n]\right)$$

括弧內可以運用每 p 個跳一次的方式加速，也就是

$$\sum_{p=1}^{N}[p\text{ 是質數}]\sum_{i=1}^{ip\leq N} f(p,ip)$$

```
for (int p = 2; p <= N; ++p) {
  if (isPrime(p)) {
    for (int i = 1; (long long) i * p <= N; ++i) {
      doSomeThing(p, i * p);
    }
  }
}
```

程式碼 1.3: 交換迴圈的加速效果

注意到這兩次 doSomeThing(p_i, n_i) 中各別 p_i, n_i 的執行次數一樣，只差在執行順序不同，如果有特別的執行順序，可以先丟到某個陣列或是 std::vector 再排序，最後至多也只有 $\mathcal{O}(N \lg N)$ 個元素。

質因數個數估計	證明題
a 是大於一的正整數，令 $f(a)$ 代表 a 的質因數個數，證明 $f(a) \le \lg a$（這個性質可以幫估在 N 個數字中 $f(a)$ 總和最高只會達到 $\mathcal{O}(n \lg M), a \le M$）	

我們可以算算看 doSomeThing 這個函數被呼叫了幾次：

$$\frac{N}{2} + \frac{N}{3} + \frac{N}{5} + \ldots \frac{N}{p} = \mathcal{O}(N \ln \ln N)$$

測出這個上界是因為尤拉老先生證出

$$\sum_{i=1}^{N} \frac{1}{p_i} = \Theta(\ln \ln N)$$

至於要在常數時間求出 isPrime(p)，我們維護 Divider[N] 為 N 的最大質因數。有一種方法可以在常數時間求出 isPrime(p)，先令 Divider 的每一項都是 0，對於每一個 p 以前的質數 q，將它的倍數 n 都標記 Divider[n] = q。當搜尋到 p 時，如果 Divider[p] == 0 代表小於 p 的質數中前面沒有任何數整除 p，因此 p 是質數。然後記得將所有 p 的倍數的 Divider 都標記為 p，依此類推，結果程式碼如下：

```cpp
int Divider[N + 10] = {};
int main() {
  for (int p = 2 ; p <= N; ++p) {
    if (Divider[p] == 0) { // => p 是質數
      for (int n = p; n <= N; n += p) {
        doSomeThing(p, n) ;
        Divider[n] = p ;
      }
    }
  }
}
```

程式碼 1.4: 篩法

注意到 Divider[N] 做完以後，對於很多的詢問，每個詢問 $n \le N$ 可以拿來做很快的（$\mathcal{O}(\log n)$）因數分解，或是 $\mathcal{O}(1)$ 測試 n 是不是質數。

分解每個詢問	經典問題
給定十萬個小於 10^6 的數字，每個詢問請在 $\mathcal{O}(\lg 10^6)$ 時間內求出質因數分解。	

如果只是要記任何一個以除以 n 的質因數，用定理 1.1 讓常數小一點，在實作上可以改成：

```
#include <iostream>
typedef long long LL;
const LL N = 1e6;
LL Divider[N + 10] = {};
void sieve() {
  for (LL p = 2; p <= N ; ++p) {
    if (Divider[p] == 0) { // p 是質數
      for (LL n = p * p; n <= (LL) N; n += p) { //差別在此，小心溢位
        // 性質：執行到此時，p | H，H < n
        // 則 Divider[H] != 0 (想一下 Divider[H] 會是什麼？)
        Divider[n] = p;
      }
      Divider[p] = p;
    }
  }
}
int main() {
  sieve();
  printf("949327 is %s\n", Divider[949327] == 949327 ?
      "a prime" : "not a prime"
  );
}
```

程式碼 1.5: 使用篩法測試質因數

執行程式以前，要不要先猜猜看答案:-)

1.1.3 最大公因數與最小公倍數 GCD & LCM

定義 1.5. *對於整數 a, b 我們稱 a, b 的**最大公因數**，是最大的正整數 d 使得 $d|a$ 且 $d|b$，記作 $d = \gcd(a, b)$。當 $\gcd(a, b) = 1$ 時，我們稱 a, b **互質**。*

這是個直觀且好用的定理：

定理 1.2. $gcd(a, b) \neq 1 \Leftrightarrow \exists p$ *是質數使得* $p|a$ *且* $p|b$

最小字典序	經典問題
一個陣列有十萬個小於 10^6 的數字。如果兩個相鄰的整數互質則可以交換，否則不行。請問這個陣列的最小的字典序是什麼？	

1.1.4 輾轉相除法 Euclid Algorithm

定理 1.3. $\gcd(a, b) = \gcd(b \pmod a, a)$

相信大多數人都在國小學過輾轉相除法了，因為用定理 1.3 遞迴寫就可以把問題慢慢變小，直到一邊整除另一邊馬上返回答案，不過現在要介紹輾轉相除法的擴充算法。

定理 1.4. $a, b, r \in \mathbb{Z}$ *若* $d := \gcd(a, b)$ *，則找得到* $s, t \in \mathbb{Z}$ *使得* $as + bt = r \Leftrightarrow d|r$

Proof. 可以透過輾轉相除法倒推構造 s, t 使得 $as + bt = d$。 □

關於如何倒推，我們注意到對於整數 a, b 可以寫成 $r = a - qb$，然後將問題慢慢變小，就請看看下面的程式碼吧！

```
#include <utility>
using namespace std;

typedef pair<int , int> ii;
ii exd(int a, int b) { // 回傳 (s, t)
  if (a % b == 0) return ii(0, 1);
  ii T = exd(b, a % b);
  return ii(T.second, T.first - a / b * T.second);
}

int main() {
  int a = 14, b = 8;
  ii ans = exd(a, b);
  printf("gcd(%d, %d) = %d = %d * %d + %d * %d\n", a, b,
    ans.first * a + ans.second * b, ans.first, a, ans.second, b);
}
// gcd(14, 8) = 2 = -1 * 14 + 2 * 8
```

程式碼 1.6: 遞迴的擴充輾轉相除法

引理 1.5 (歐幾里德引理)**.** *已知* $\gcd(a, b) = 1$ *，那麼若* $a|bc$ *，則* $a|c$

Proof. 存在 s, t 使得 $as + bt = 1$，兩邊同乘以 c 得到 $acs + bct = c \Rightarrow a|c$ □

1.2 模運算 Modular Arithmetics

1.2.1 同餘式與模數 Congruence & Modulo

定義 1.6. *若* $n|(a - b)$ *，則記作* $a \equiv b \pmod{n}$

定理 1.6. *以下是幾個模數運算常見的定理：*

1. $a \equiv b \pmod{n},\ c \equiv d \pmod{n} \Rightarrow a + c \equiv b + d \pmod{n}$

2. $a \equiv b \pmod{n}$, $c \equiv d \pmod{n} \Rightarrow ac \equiv bd \pmod{n}$

3. $a \equiv b \pmod{n} \Rightarrow a^k \equiv b^k \pmod{n} \forall k \in \mathbb{N}$

這裡挑第三點來證：

Proof.

$$n|(a-b) \Rightarrow n|(a-b)(a^{k-1} + a^{k-2}b + a^{k-3}b^2 + \ldots + ab^{k-2} + b^{k-1}) \Rightarrow n|(a^k - b^k)$$

□

利用模數運算，我們可以定義出一種代數結構：

定義 1.7. *對於 n 大於 1，定義 $\langle \mathbb{Z}_n, +_n, \times_n \rangle$，簡寫為 $\mathbb{Z}_n$，其中：*

$$\mathbb{Z}_n = \{0, 1, 2, 3, 4, \ldots, n-1\}$$

$$a +_n b := (a+b) \pmod{n},\ a \times_n b := (a \times b) \pmod{n} \forall a, b \in \mathbb{Z}_n$$

沒錯！這就是平常熟悉的% 運算，不過請注意正負值。

定理 1.7. *在 $\mathbb{Z}_n$ 運算下同樣滿足*

1. *加法乘法結合率：*

$$(a \times_n b) \times_n c = a \times_n (b \times_n c),\ (a +_n b) +_n c = a +_n (b +_n c)$$

2. *加法乘法交換率：*

$$a \times_n b = a \times_n b,\ a +_n b = b +_n a$$

3. *分配律：*

$$(a +_n b) \times_n c = a \times_n c +_n b \times_n c$$

有時兩個不同模的數乘以常數後會是同模的，這時有消去定理，同時也給出在模運算之下「除法」的概念：

定理 1.8 **(消去定理).** $d = \gcd(c, n)$

$$ca \equiv cb \pmod{n} \Rightarrow a \equiv b \pmod{\frac{n}{d}}$$

Proof. 因為有定理 1.5,

$$n|c(a-b) \Rightarrow \frac{n}{d}\bigg|\frac{c}{d}(a-b)$$

而且

$$\gcd\left(\frac{n}{d}, \frac{c}{d}\right) = 1$$

我們可以推到

$$\frac{n}{d}\bigg|(a-b) \Rightarrow a \equiv b \pmod{\frac{n}{d}}$$

□

1.2.2 反元素 Inverse Element

定義 1.8. *$a, b \in \mathbb{Z}_n$ 我們稱 b 是 a 的**乘法反元素**（簡稱反元素），有*

$$a \times_n b = 1 \Leftrightarrow b := a^{-1},\ a := b^{-1}$$

$\mathbb{Z}_n$ 中，不是每個數都有反元素。例如，0 在任何 $\mathbb{Z}_n$ 一定沒有反元素；$\mathbb{Z}_4$ 中 2 沒有反元素。以 $\mathbb{Z}_9$ 為例，看看它的反元素是什麼：

a	0	1	2	3	4	5	6	7	8
a^{-1}	無	1	5	無	7	2	無	4	8

表 1.1: $\mathbb{Z}_9$ 乘法反元素表

由此可以觀察可發現以下定理：

定理 1.9. *a 在 $\mathbb{Z}_n$ 中有乘法反元素若且唯若 $\gcd(a, n) = 1$。*

Proof. 輾轉相除擴充演算法總是能把反元素造出來，可以利用擴充算法找模 n 下的反元素：

$$ax + py = 1 \Leftrightarrow ax = 1 - py \Leftrightarrow ax \equiv 1 \pmod{p} \Leftrightarrow x \equiv a^{-1} \pmod{p}$$

□

輾轉相除擴充演算法分解的步驟很少，寫成迭代更快！

定理 1.10. *反元素是唯一的，也就是說如果 $a \times_n b = 1,\ a \times_n c = 1$ 則 $b \equiv c \pmod{n}$*

Proof. $a \times_n (b - c) = 0$，也就是 $n|a(b-c)$ 。
因為 $\gcd(n, a) = 1$，利用定理 1.5 可以推得 $n|(b-c)$ □

1.2.3 費馬小定理 Fermat's Little Theorem

定理 1.11 (費馬小定理). *對於任何整數* a *，質數* p *皆滿足* $a^p - a \equiv 0 \pmod{p}$ *。若* a *不是* p *的倍數，則可以推得* $a^{p-1} \equiv 1 \pmod{p}$

Proof. 費馬有一個很關鍵的發現：對於任何 $a \in \mathbb{Z}_n,\ a \neq 0$，在

$$1a, 2a, 3a, 4a, \ldots, (p-1)a$$

中必須一一對應到

$$1, 2, 3, 4, \ldots, (p-1)$$

否則，會造成矛盾：$i > j,\ ia \equiv ja \pmod{p} \Rightarrow p|(i-j)a$。

$$1a, 2a, 3a, 4a, \ldots, (p-1)a,\ 1, 2, 3, 4, \ldots, (p-1)$$

在兩邊連乘得到：

$$(p-1)!a^{p-1} = a \cdot 2a \cdot 3a \cdots (p-1)a \equiv 1 \cdot 2 \cdot 3 \cdots (p-1) = (p-1)! \pmod{p}$$

根據定理 1.8，兩邊消去 $(p-1)!$ 得到 $a^{p-1} \equiv 1 \pmod{p}$
這說明了本節的重點：任何一個非 p 倍數的正整數，都有乘法反元素 a^{p-2}（還記得定理 1.10 說明反元素是唯一的嗎？) □

現在我們知道 $\mathbb{Z}_p$ 的元素支援加減，非零元素支援乘除。像這樣的性質的結構，數學家們稱作是一個**體**（Field）。

因此可以使用快速冪來求 a^{p-2}：

```
typedef long long LL;
LL pow(LL a, LL p, LL mod) { // 也可以寫寫看遞迴版本
  if (a % mod == 0) return 0;
  LL ret = 1ll % mod;
  for (LL cur = a; p; cur = cur * cur % mod, p >>= 1) {
    if (p & 1ll) {
      ret = ret * cur % mod;
    }
  }
  return ret;
}
int main() {
  const LL prime = 7;
  for (LL a = 1, inva; a < prime; ++a) {
    inva = pow(a, prime - 2, prime);
    printf("a = %2lld, a^-1 = a^(p-2) = %2lld, a * a^(p-2) = %lld\n",
        a, inva, a * inva % prime);
```

```
  }
}

/*
a =  1, a^-1 = a^(p-2) =  1, a * a^(p-2) = 1
a =  2, a^-1 = a^(p-2) =  4, a * a^(p-2) = 1
a =  3, a^-1 = a^(p-2) =  5, a * a^(p-2) = 1
a =  4, a^-1 = a^(p-2) =  2, a * a^(p-2) = 1
a =  5, a^-1 = a^(p-2) =  3, a * a^(p-2) = 1
a =  6, a^-1 = a^(p-2) =  6, a * a^(p-2) = 1
*/
```

程式碼 1.7: 快速冪求反元素

1.2.4 歐拉函數 Euler Function

定義 1.9. ***歐拉函數*** *$\Phi(n)$ 的值表示在 $1, 2, \ldots, n$ 中與 n 互質的個數。*

我們使用 $\mathbb{Z}_n^*$ 代表從 $[0, n)$ 中與 n 互質的數，也可以說 $\mathbb{Z}_n^*$ 有 $\Phi(n)$ 個元素。

定理 1.12 (Euler)**.** *如果 $\gcd(a, n) = 1$，則 $a^{\Phi(n)} \equiv 1 \pmod n$*

Proof. 注意到費馬小定理的證法，如果 $\gcd(a, n) = 1$，也可以用來證明 $a^{\Phi(n)} \equiv 1 \pmod n$ □

定理 1.13. *若 $\gcd(m, n) = 1$，則 $\Phi(mn) = \Phi(m)\Phi(n)$*

Proof. 我們可以注意下圖，可以發現只有 $\Phi(m)$ 個列有跟 m 互質的數（為什麼呢？），每個列剛好有 $\Phi(n)$ 個項跟 n 互質（因為列中的元素會跟 $0, 1, 2, \ldots, n-1 \pmod n$ 一一對應） □

$$
\begin{array}{ccccc}
1 & 2 & 3 & \ldots & m \\
m+1 & m+2 & m+3 & \ldots & 2m \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
(n-1)m+1 & (n-1)m+2 & (n-1)m+2 & \ldots & nm
\end{array}
$$

定理 1.14.

$$\sum_{d|n} \Phi(d) = n$$

Proof. 若 $d|n$，在 $\{1, 2, 3, 4, 5, \ldots, n\}$ 裡面總共有 $\Phi(d)$ 個元素跟 n 的最大公因數是 $\frac{n}{d}$ 則 $\sum_{\frac{n}{d}|n} |\{a|gcd(a, n) = \frac{n}{d}\}| = \sum_{\frac{n}{d}|n} \Phi(d) = \sum_{d|n} \Phi(d) = n$ □

請從上面性質自行推出歐拉函數公式：

歐拉函數公式	證明題
給予一個 $n = p_1^{k_1} p_2^{k_2} p_3^{k_3} \cdots p_n^{k_n}$，請算出 $\Phi(n)$。	

1.2.5 數論函數 Number-Theoretic function

定義 1.10. *一個數論函數 $f(n)$ 是一個定義在正整數的函數，值域是實數（或複數）。*

簡單來說數論函數吃一個正整數，吐出一個實數或複數。

定義 1.11. *設數論函數 $f(n)$，若 $\gcd(m, n) = 1$，滿足 $f(1) = 1$ 且 $f(mn) = f(m)f(n)$ 的話我們稱 f 為**積性函數** (Multiplicative function)*

例子像是 $\Phi(x)$ 是積性函數，考慮到任何一個正整數可以分解成質數的冪次，要算積性函數 $f(p_1^{k_1} p_2^{k_2} ... p_n^{k_n})$，我們只要計算 $f(p_1^{k_1}) f(p_2^{k_2}) ... f(p_n^{k_n})$ 即可

構造積性函數	經典問題
給一堆兩兩互質的正整數 $\{a_i\}_{i=0}^{N-1}, a_i \leq 10^6$，還有 $f(a_i)$ 們也給了，請造出任何一個符合條件積性函數 f，構造方法為輸出 $f(1), f(2), \ldots, f(10^6)$，若無解請輸出 -1	

定理 1.15. *若 $f(n)$ 是積性函數，則 $F(n) = \sum_{d|n} f(d)$ 也是積性函數*

Proof. 若 m, n 互質，可以把任何 mn 的因數拆開寫成 $d_1|m, d_2|n$，$d_1 d_2|mn$

$$F(mn) = \sum_{d|nm} f(d) = \sum_{d_1|m, d_2|n} f(d_1) f(d_2) = \sum_{d_1|m} f(d_1) \sum_{d_2|n} f(d_2) = F(m)F(n)$$

□

對於數論函數，有所謂狄利克雷卷積（Dirichlet convolution）：

定義 1.12. *我們定義兩個數論函數 f, g 的**狄利克雷卷積**為*

$$(f * g)(n) := \sum_{d_1|n} f(d_1) g(\frac{n}{d_1})$$

或者寫得更直觀一點：

$$(f * g)(n) := \sum_{d_1 d_2 = n} f(d_1) g(d_2)$$

三個函數的狄利克雷卷積長這樣，依此類推

$$((f*g)*h)(n) = \sum_{d_1d_2d_3=n} f(d_1)g(d_2)h(d_3)$$

很明顯的可以看出狄利克雷卷積有結合率、交換率：

1. $(f*g)*h$
   $= \sum_{d_1d_2d_3=n} f(d_1)g(d_2)h(d_3) = f*(g*h)$

2. $f*g$
   $= \sum_{d_1d_2=n} f(d_1)g(d_2) = g*f$

以下介紹幾個等等會用到的積性函數，可以自行驗證看看它是否是積性函數

1. $I(x) := [x=1]$

2. 常數函數 1：$1(x) := 1$

3. 莫比烏斯函數：

$$\mu(x) = \begin{cases} 1 & x=1 \\ 0 & p^2|x,\ p \text{ 是質數} \\ (-1)^k & x = p_1p_2\cdots p_k \end{cases}$$

注意一下

$$\sum_{d|x} \mu(d) = [x=1] = I(x)$$

(可令 $x = p^k$ 證證看)

1.2.6 莫比烏斯反演 Möbius Inversion Formula

定理 1.16. *對於數論函數 f 跟 F 關係如下：*

$$F(x) = \sum_{d|x} f(d)$$

那麼我們有：

$$f(x) = \sum_{d|x} \mu(d) F\left(\frac{x}{d}\right)$$

Proof. 核心的證明想法就是交換 $\sum$ 的技巧，這在比賽中很常見

$$\sum_{d|x}\mu(d)F\left(\frac{x}{d}\right)=\sum_{d|x}\mu(d)\sum_{c|\frac{x}{d}}f(c)=\sum_{d|x}\sum_{c\,|\frac{x}{d}}f(c)\mu(d)$$

改成這個寫法好看多了

$$\sum_{cd|x}f(c)\mu(d)$$

拜託拜託，一定要注意到

$$(d|x\ \text{且}c|\frac{x}{d})\Leftrightarrow cd|x\Leftrightarrow(c|x\ \text{且}d|\frac{x}{c})$$

因此原式可以寫回

$$\sum_{c|x}\sum_{d|\frac{x}{c}}f(c)\mu(d)=\sum_{c|x}f(c)\sum_{d|\frac{c}{d}}\mu(d)=\sum_{c|x}f(c)[x=c]=f(x)$$

□

或者大可以把剛剛那一段醜醜的證明忘記，改成看這個簡潔有力的證明：

Proof.

$$\sum_{d|x}\mu(d)F(\frac{x}{d})=(\mu*f*1)(x)=((\mu*1)*f)(x)=([x=1]*f)(x)=\sum_{d|x}[d=1]*f(\frac{n}{d})=f(x)$$

□

The Holmes Children	CF 776E
f 是歐拉函數，$g(n)=\sum_{d\|n}f(\frac{n}{d})$，試著計算 $$F_k(n)=\begin{cases}f(g(n)) & k=1\\ g(F_{k-1}(n)) & k>1,\ k\text{ 是偶數}\\ f(F_{k-1}(n)) & k>1,\ k\text{ 是奇數}\end{cases}$$ 將答案模 1000000007 輸出	

1.2.7 原根 Primitive Root

定義 1.13. *對於質數 p ，我們說 $a\in\mathbb{Z}_p^*$ 的**序**（order）是最小的正整數 d，使得 $a^d=1$*

這樣講可能有點抽象，舉個例子：
12 在 $\mathbb{Z}_{29}^*$ 的序是 4，因為 $12^4 \equiv 1 \pmod{29}$，而且比 4 小的正整數次方都不是 1
5 在 $\mathbb{Z}_{101}^*$ 的序是 25，因為 $5^{25} \equiv 1 \pmod{101}$，而且比 25 小的正整數次方都不是 1

定義 1.14. *對於質數 p，我們說 $a \in \mathbb{Z}_p^*$ 是 $\mathbb{Z}_p^*$ 的**原根**（Primitive Root，群論用語會翻 generator），如果比 $p-1$ 小的 a 的正整數冪次都不是 1，也就是說，a 的序是 $p-1$。*

可以證明這樣的 a 一定存在，證明這裡先沒寫出來。

定理 1.17. *若 a 是 $\mathbb{Z}_p^*$ 的原根，則 $a^0, a^1, a^2, \ldots, a^{p-2}$ 一一對應到 $1, 2, 3, 4, \ldots, p-1$*

Proof. 若 $0 \le i < j < p-2$, $a^i = a^j$，則依據消去定理 $a^{j-i} \equiv 1 \pmod{p}$ 不符合定義 □

例如 3 是 $\mathbb{Z}_7^*$ 的原根：

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$$

而 3 的冪次表如下

t	1	2	3	4	5	6
3^t	3	2	6	4	5	1

表 1.2: 3 的冪次表

定理 1.18. *正整數 $d|p-1$，$\mathbb{Z}_p^*$ 中序為 d 的元素剛好有 $\Phi(d)$ 個*

Proof. 那些元素就是 $\{a^k : gcd(k, p-1) = \frac{p-1}{d}\}$ 中的元素，a 是任一個原根，至於為什麼它們的序是 d：
$(a^k)^x = 1 \Rightarrow p-1|kx$ 再利用引理 1.5 可以推到 $d|x$ □

給一個質數 p，我們有辦法找到任一個原根嗎？我們當然可以一個個試 $2, 3, 4, \ldots, p-1$，然後將它們從 $1, \ldots, p-1$ 次方都算過一次，但若 $p \le 10^9$，可能就沒有那麼容易了，關於這個問題，有一種機率解法是這樣：

演算法 1.1 找一個原根的演算法

1: 先把 $p-1$ 作因式分解，$p-1 = \prod_{i=1}^r q_i^{e_i}$
2: 對於每一種 q_i 隨便挑一個數 α_i 使得 $\alpha_i^{\frac{p-1}{q_i}} \ne 1$，令 $\gamma_i = \alpha_i^{\frac{p-1}{q_i^{e_i}}}$，失敗了就再來一遍
3: 輸出答案 $\gamma = \prod_{i=1}^r \gamma_i$

在每一步驟中，簡單說明一下為什麼要這樣做：

1. 第二步，如果這樣挑可以令 $\gamma_i^{q_i^{e_i-1}} \neq 1$ ，且 $\gamma_i^{q_i^{e_i}} = 1$ 。而挑中的機率其實挺大的，假設 a 是原根，只要挑中的不是 $a^{q_i}, a^{2q_i}, a^{3q_i}, ..., a^{p-1} = 1$ 即可，平均做兩次以內就找得到。在此，γ_i 的序是 q^{e_i} 。

2. 第三步，某個理論（群論）告訴我們：如果 a, b 的序互質，則 ab 的序就是各自的序的乘積，因此 γ 的序是 $p-1$ ，講白一點，γ 就是原根。

2. 3. 步驟複雜度的期望值是 $\mathcal{O}(r \log p)$，r 是質因數個數，已經幾乎是常數了，因此最慢的部分還是在分解 $p-1$

現在問題反過來，給一個數 γ ，它是不是 $\mathbb{Z}_p^*$ 的原根？如果是的話，假設 a 是另一個原根，則 $\gamma = a^d$，$\gcd(p-1, d) = 1$，否則一定有個質數 q 使得 $q|gcd(p-1, d)$，則 $p-1|\frac{p-1}{q}d$，我們只要一一測試分解 $p-1$ 的質數，看看 $\gamma^{\frac{p-1}{q}}$ 是否等於一就好了。

以下附上程式碼，主函式分為兩個部分，第一部分是把一百萬以內的質數用篩法找出來，第二部分，對於這些質數，利用上述方法找出它們的原根，再來是測試原根的驗證程式：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long LL;
vector<LL > Qs;

#define MAXN 1000001
LL Divider[MAXN] = {};
void sieve(LL N = MAXN) {
  for (LL p = 2; p < MAXN; ++p) {
    if (Divider[p] == 0) {
      for (LL n = p; n < MAXN; n += p) {
        Divider[n] = p;
      }
    }
  }
}
LL modPow(LL a, LL power, LL mod) {
  LL ans = 1;
  for (LL cur = a; power; power >>= 1, cur = cur * cur % mod) {
    if (power & 1) ans = ans * cur % mod;
  }
  return ans;
}
void factor(LL N) { // 把分解p-1的質數都找出來
  while (N != 1) {
    Qs.push_back(Divider[N]);
    while (Divider[N] == Qs.back())
      N /= Divider[N];
  }
```

```
}
LL root(LL prime) { // 找原根
  Qs.clear();
  factor(prime - 1);
  LL gamma = 1;
  for (LL q_i : Qs) { // Range-based for loop since C++11
    LL alpha_i = 1, b ,N = prime - 1;
    while (N % q_i == 0) N /= q_i;
    do {
      ++alpha_i; // 沒錯，這就是我的隨機函數！
      b = modPow(alpha_i, (prime - 1) / q_i, prime);
    } while(b == 1);
    gamma = gamma * modPow(alpha_i, N, prime) % prime;
  }
  return gamma;
}
// 測試 a 是否為原根
LL isPrimitiveRoot(LL a, LL prime, vector<LL > &Qs) {
  for (LL q : Qs) {
    if (modPow(a, (prime - 1) / q, prime) == 1)
      return false;
  }
  return true;
}
int main() {
  sieve();//篩法
  for (LL p = 2; p < MAXN ; ++p) {
    if (Divider[p] == p) { // p 是質數
      LL a = root(p); // 找一個原根
      if (isPrimitiveRoot(a, p, Qs))
        printf("%6lld has order %6lld under module %6lld\n",
          a, p - 1, p);
      else puts("test fail.");
    }
  }
}
```

程式碼 1.8: 尋找與測試原根

1.2.8 中國剩餘定理 Chinese Remainder Theorem

定理 1.19 (中國剩餘定理). *令 $\{n_k\}_{i=1}^{k}$ 為兩兩互質的正整數，令 $a_1, a_2, ..., a_k$ 為任意的整數，則存在 $a \in \mathbb{Z}$ 使得滿足：$a \equiv a_i \pmod{n_i}$，$i = 1, \cdots, k$ 再者，令 $n = \prod_{i=1}^{k} n_i$，$a' \in \mathbb{Z}$ 也是一個解若且唯若 $a \equiv a' \pmod{n}$*

Proof. 如果我們可以知道 $e_i \equiv \begin{cases} 1 \pmod{n_i} \\ 0 \pmod{n_j} & j \neq i \end{cases}$，則可以造出 $a \equiv \sum_{i=1}^{k} a_i e_i \pmod{n}$ 要如何找出 e_i 呢？用歐幾里德擴充算法：

令 $n_i^* = \frac{n}{n_i} = \prod_{i=1, i\neq j}^{k} n_i$，有 $sn_i + tn_i^* = 1$，則 $e_i := tn_i^*$ 就完成了 □

```
#include <bits/stdc++.h>
#define MAXN 101
using namespace std;
typedef long long LL;
typedef pair< LL , LL > ii;

ii exd_gcd( LL a , LL b ){//return s,t
  if( a % b == 0 ) return ii( 0 , 1 );
  ii T = exd_gcd( b , a % b );
  return ii( T.second , T.first - a / b * T.second );
}
LL a[ MAXN ] = { 3, 8, 6 }, n[ MAXN ] = { 17, 13, 15 }, e[ MAXN ], k = 3;

int main(){
  LL prodN = 1, ans = 0, nStar;
  for(LL i = 0 ; i < k ; ++ i ) prodN *= n[ i ];
  for(LL i = 0 ; i < k ; ++ i )
    nStar = prodN / n[i], e[i] = exd_gcd( n[ i ], nStar).second * nStar;
  for(LL i = 0 ; i < k ; ++ i ) (ans += e[ i ] * a[ i ] % prodN) %= prodN
      ;
  printf( "%lld\n" , ( ans + prodN ) % prodN );
}
```

程式碼 1.9: 測試中國剩餘定理

1.3 快速傅立葉變換 Fast Fourier Transform

1.3.1 生成函數 Generating Function

生成函數是一種多項式的衍生函數 $F(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \ldots$，函數行為意義不大，主要用途是對於用來查他的第 N 次項，可以把它想像成是有無限多項的陣列。像是 $G(x) = 1 + \frac{1}{2}x + \frac{1}{4}x^2 + \frac{1}{8}x^3 + \frac{1}{16}x^4 \ldots$ 可以想像成 $< 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64} \ldots >$

定義 1.15. *我們定義一個序列* $< a_i >_{i=0}^{\infty}$ *的生成函數* $F(x)$ *可被寫作* $F(x) = \sum_{i=0}^{\infty} a_i x^i$ *對於一個生成函數，我們用* $[x^n]F(x)$ *來代表* $F(x)$ *的第* n *次項*

相加（減）相乘就跟多項式一樣

定義 1.16. 令

$$A(x)=\sum_{i=0}^{\infty}a_ix^i, B(x)=\sum_{i=0}^{\infty}b_ix^i$$

兩個生成函數的相加定義為

$$A(x)+B(x)=\sum_{i=0}^{\infty}(a_i+b_i)x^i$$

兩個生成函數的相乘定義為

$$[x^n](AB)(x)=\sum_{i=0}^{n}\sum_{j=0}^{n}a_ib_j[i+j=n]=\sum_{i=0}^{n}a_ib_{n-i}=\sum_{i=0}^{n}a_{n-i}b_i$$

生成函數與 dp 還可以扯得上邊，甚至是拿來分析 dp 的好工具呢
如果遇到像是 $G(x)=1+kx+k^2x^2+k^3x^3+k^4x^4...$ 的生成函數，令 $A(x)=\sum_{i=0}^{\infty}a_ix^i$，計算 $[x^n](GA)(x)$ 時，不必花 $\mathcal{O}(n^2)$ 將他們相乘，留意到

$$[x^n]\left(GA\right)(x)=\sum_{i=0}^{n}k^{n-i}a_i=a_n+k\sum_{i=0}^{n-1}k^{n-1-i}a_i=a_n+k[x^{n-1}]\left(GA\right)(x)$$

變成了漂亮的 dp 式。

01 背包計數問題	經典問題
有一個小偷有容量 V 的背包，考慮 N 個物品，每個物品的體積不同，但我們今天不問小偷能不能塞滿這個背包，我們想知道它有多少種方法把這個背包塞滿，請在 $\mathcal{O}(NV)$ 時間內對於每種 V 都輸出該答案，如果今天方法 A 所塞的物品，方法 B 都有了，而且方法 B 所塞的物品，方法 A 都有了，那我們就說方法 A 跟方法 B 是一樣的，不然這兩種方法就是不一樣的	

令第 i 個物品重量為 w_i，那答案就是 $[x^V]\prod_{i=0}^{N-1}(1+x^{w_i})$ 這其實就是 dp 式的表現，令`dp[i][v]`為前 i 個選項中，容量為 v 的答案：則`dp[i][v]=dp[i-1][v-w_i]`+`dp[i-1][v]`對應到的生成函數是

$$\prod_{j=0}^{i-1}(1+x^{w_j})\times(1+x^{w_i})$$

定理 1.20. *一個生成函數函數乘上* $(1+x+x^2+x^3+x^4+...)$*，會變成它本身的前綴和*

考慮到幾何級數，有時候我們會把生成函數 $(1+x+x^2+x^3+x^4+...)$ 寫成 $\frac{1}{1-x}$（因為在 x 絕對值小於一時兩邊是一樣的）想要把 I 個區間都加 k_i，而題目只有在最後做 query，利用生成函數的思想，不必用線段樹就可以把複雜度做到 $O\left(N+I\right)$，簡單又好寫：留意到

$$(x^n + x^{n+1} + x^{n+2} + ...x^m) = (1 + x + x^2 + x^3 + x^4 + ...)(x^n - x^{m+1})$$

$$k_1(x^{n_1} + x^{n_1+1} + ...x^{m_1}) + k_2(x^{n_2} + x^{n_2+1} + ...x^{m_2}) + ...$$

$$:= \sum_{i=1}^{I} k_i \sum_{j=n_i}^{m_i} x^j = \sum_{i=1}^{I} k_i \frac{(x^n - x^{m+1})}{1-x} = \frac{1}{1-x} \sum_{i=1}^{I} k_i (x^n - x^{m+1})$$

告訴我們可以先在陣列 n_i, m_{i+1} 個別加上 $k_i, -k_i$ ，最後再做前綴和即得到答案

二項式恆等式	證明題
利用生成函數證明： $$\sum_{i=0}^{r} \binom{n}{i}\binom{m}{r-i} = \binom{m+n}{r}$$	

那為什麼要提到生成函數呢，它的功用主要在於計數分析，不過有一部分計數的問題需要用到生成函數乘積，這時候一個好的多項式乘法 algo 會變得相當重要，因為要算 n, m 次多項式 $P(x), Q(x)$ 相乘的時候，如果沒有特別好的條件的話，生成函數相乘硬解的複雜度會是 $\mathcal{O}(n^2)$，但假設現在我們知道了 $\{(x_i, P(x_i)Q(x_i))\}_{i=1}^{n+m+1}$，利用等會兒提到的多項式插值唯一定理，可以找回 $P(x)Q(x)$

1.3.2 離散傅立葉變換 Discrete Fourier Transform

在進入離散傅立葉變換（DFT）之前，先看個引理吧

引理 1.21 (多項式插值唯一定理 Uniqueness of an interpolating polynomial)**.** *一個平面上一堆點 $\{(x_i, y_i)\}_{i=0}^{n-1}$，$x_i$ 們各不相同，存在唯一一個多項式 $P(x)$ 使得 $P(x_i) = y_i$，使得多項式最高次項小於 n*

Proof. 假設我們想要知道的多項式是 $P(z) = a_0 + a_1 z + a_2 z^2 + ... + a_{n-1} z^{n-1}$ 問題就變成了

1. $a_0, a_1, ..., a_{n-1}$ 是什麼

2. 解為什麼是唯一的

我們有恆等式：

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(x_0) \\ P(x_1) \\ P(x_2) \\ \vdots \\ P(x_{n-1}) \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

左邊那個矩陣，我們叫它范德蒙矩陣（Vandermonde matrix）注意到上式中，y_i, x_i 我們已經有了，看來我們還需要擅長取反矩陣找 a_i 的朋友呢

等等，這矩陣是可逆的嗎？

可以配合歸納法來證明：

$$\det \begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} = \prod_{0 \le i < j < n} (x_j - x_i)$$

因為 x_i 們是各不相同，這個矩陣的行列式 (det) 不為零，當然是可逆的，而且這同時也告訴我們 $a_0, a_1, ..., a_{n-1}$ 是唯一決定的！ □

高斯消去法取反矩陣要花 $\mathcal{O}(n^3)$，拉格朗日插值法要花 $\mathcal{O}(n^2)$，然而，如果 x_i 選得很特別，有辦法做到 $\mathcal{O}(n \lg n)$ 時間在 a_i, y_i 間快速轉換，這個我們叫它 FFT，在介紹 FFT 以前，先從 DFT 來下手。

假設有一種數字 x ，對於 n 它滿足 $\sum_{i=0}^{n-1} x^i = 0$ ，除了零以外真的有這種鬼數字存在嗎？有的！
考慮對於 $nd+1$ 的質數 p，討論 $\mathbb{Z}_p^*$ 時，這種鬼數字會存在，或者是不真實（real）而複雜（complex）的數，我們叫它複數 $\mathbb{C}$。

先說第一種情況：

定理 1.22. *若 p 是一個 $nd+1$ 的質數，a 是一個原根，令 $b = a^d$ ，則 b 的序為 n*

Proof. 令 x 為 b 的序
若 x 比 n 小，則 $b^x = a^{xd} = 1$，則 a 不是原根，因為它的序比 nd 小，矛盾。
而 $b^n = a^{nd} = a^{p-1} = 1$ 表示 b 的序比 n 小或等於 n，因此 b 的序是 n □

在此 $\sum_{i=0}^{n-1} x^i = 0$ 的非零解是什麼呢？答案是在模 p 下的 $b^k, k = 1, 2, ...n-1$
為了標記方便，對於 $\mathbb{Z}_p^*$ 的元素，我們在這章節統稱共軛為反元素，以 $\bar{b} = b^{-1}$ 表示。

對於第二種情形，或許已經有人知道複數的概念，但這裡再提一下，複數是一個平面 $\mathbb{C} = \mathbb{R} + i\mathbb{R}$，其中 $i \times i = -1$，複數支援加減乘除，若 $A = a_1 + ia_2, B = b_1 + ib_2$，則

1. 我們稱 A 的共軛複數為 $\overline{A} = a_1 - ia_2$
2. $A + B = (a_1 + b_1) + i(a_2 + b_2)$
3. $A - B = (a_1 - b_1) + i(a_2 - b_2)$
4. $A \times B = (a_1 b_1 - a_2 b_2) + i(a_1 b_2 + a_2 b_1)$
5. $A \div B = \frac{a_1 + ia_2}{b_1 + ib_2} = \frac{(a_1 + ia_2)(b_1 - ib_2)}{(b_1 - ib_2)(b_1 + ib_2)} = \frac{(a_1 b_1 + a_2 b_2) + i(a_2 b_1 - a_1 b_2)}{b_1^2 + b_2^2}$

物件 `std::complex` 支援這些運算，包括三角函數的使用。

可以用 xy 座標畫出複數的點，一個複數的絕對值為它到原點 $0 + i0$ 的歐幾里德距離（用尺量出來的那個距離）

這些數字都不是自然存在的數，你的午餐價格不可能是 $3 + 2i$，但就是因為複數有很好用的結構（複結構），才會被發展出來做解方程式的根、FFT 之類的事

在此 $\sum_{i=0}^{n-1} x^i = 0$ 的非零解是什麼呢？答案是 $e^{\frac{2\pi ik}{n}}, k = 1, 2, ...n-1$，這些數的共軛跟反元素是一樣的
我們有歐拉公式 Euler Formula：$e^{ix} = \cos x + i \sin x$
這是展開式 $e^x = \frac{1}{0!} + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ...$ 在複平面上推廣所得出的結果。

如果是 $\mathbb{Z}_p^*$ 我們令 $w_N^k = b^k$，如果是 $\mathbb{C}$ 上我們令 $w_N^k = e^{2\pi i \frac{k}{N}}$（說穿了就只是將 1 對著原點逆時針旋轉 $\frac{k}{N}$ 個圓周的複數，如果 $k < 0$ 就代表順時針旋轉 $\frac{|k|}{N}$ 個圓周），為了直觀我們就用圓餅圖示範（下表），這裡我們叫它「派運算」，派運算表示時為了方便起見 N 都是 8，值得一提的是，派運算不管是對於 $\mathbb{Z}_p^*$ 上的 $w_N = b$，還是 $\mathbb{C}$ 上的 $w_N = e^{\frac{2\pi i}{N}}$，下面敘述都是滿足的，因此講者以下不再區分兩者的差別：

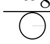
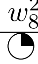
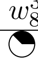
w_8^0	w_8^1	w_8^2	w_8^3	w_8^4	w_8^5	w_8^6	w_8^7
 或 							

表 1.3: 派-複數根對照表

定理 1.23. w_N^i *與派運算有以下規則*

1. $w_N^i \times w_N^j = w_N^{i+j}$，*在派運算下就是將面積相加，例如* $=$

2. $(w_N^i)^j = w_N^{ij}$，*在派運算下就是將面積相乘上一個常數，例如* $^3 =$

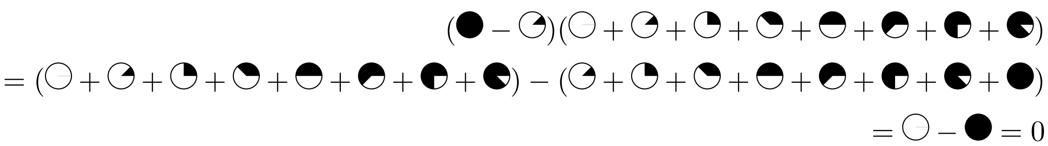
3. w_N^{-i} *是* w_N^i *的共軛，在派運算下就是將一塊完整的派拿走原來派的面積，例如* $=$

4. $w_N^i = w_N^{i+N}$，*在派運算下就是將面積模一塊完整的派，例如* $=$

5. $w_{Nk}^{ik} = w_N^i$，*在派運算下就是指* *切成四塊或是八塊面積都是一樣的：* $= =$

6. $\sum_{k=0}^{N-1}(w_N^i)^k = N[N|i]$，*在派運算下就像是* $\sum_{k=0}^{N-1}(\ )^k = 0$

Proof. 我們證最後一點的派式子，若 $i \neq 0 \pmod N$，可以發現：

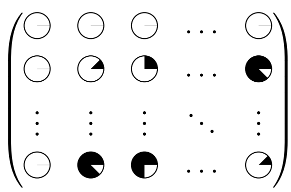


因此 $= 0$ □

注意 3 $\neq$ 而是應該要像向量一樣直接寫成 3

定義 1.17. $w_N^0, w_N^1, ...w_N^{N-1}$ *所構成的范德蒙矩陣，我們叫它 DFT 矩陣，以* D_N *簡寫*

$$D_N = \begin{pmatrix} 1 & w_N^0 & w_N^{0*2} & \cdots & w_N^{0*(N-1)} \\ 1 & w_N^1 & w_N^{1*2} & \cdots & w_N^{1*(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_N^{N-1} & w_N^{(N-1)*2} & \cdots & w_N^{(N-1)*(N-1)} \end{pmatrix} =$$



是個漂亮的對稱矩陣呢！

它的反矩陣長這樣：

$$D_N^{-1}=\frac{1}{N}\begin{pmatrix}1 & w_N^{-0} & w_N^{-0*2} & \cdots & w_N^{-0*(N-1)}\\ 1 & w_N^{-1} & w_N^{-1*2} & \cdots & w_N^{-1*(N-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & w_N^{1-N} & w_N^{(1-N)*2} & \cdots & w_N^{(1-N)*(N-1)}\end{pmatrix}=\frac{1}{N}\begin{pmatrix}\text{●} & \text{●} & \text{●} & \cdots & \text{●}\\ \text{●} & \text{◕} & \text{◕} & \cdots & \text{◔}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \text{●} & \text{◔} & \text{◑} & \cdots & \text{◕}\end{pmatrix}$$

簡單來說 D_n 的反矩陣是 D_n 每個項取共軛再除以 N，如果學員學過（或未來會學到）線性代數，就可以知道這是一個 Unitary Matrix（忽略常數），自己乘乘看是不是單位矩陣吧。

從上述的定理我們可以知道一個小於 n 次的多項式的呈現，除了可以用 $P(z)=a_0+a_1z+a_2z^2+...+a_{n-1}z^{n-1}$ 表示，還可以用 n 個點來代表一個唯一的多項式，這個表示法叫做點值表示法（Point Value Representation）

發現了嗎，如果有兩個多項式的點值表示法，而它們選用的那些 x_i 都一樣，要怎麼得到兩個多項式相乘的點值式呢？直接把每個 y 座標都相乘就好了對吧。

演算法 1.2 離散傅立葉變換

1: 先用 D_N 矩陣乘積將 $P(\text{○}), P(\text{◔}), P(\text{◑}), ..., P(\text{◕})$，$Q(\text{○}), Q(\text{◔}), Q(\text{◑}), ..., Q(\text{◕})$ 算出來。
2: 對於每個 ◔^i 直接相乘兩個點值表示法的多項式 $P(\text{○})Q(\text{○})$，$P(\text{◔})Q(\text{◔})$，...，$P(\text{◕})Q(\text{◕})$
3: 用反矩陣 D_N^{-1} 將將 $P(x)Q(x)$ 的 x^i 每一項算出

這個算法是 $\mathcal{O}(n^2)$，門檻在 1. 3. 步驟，但待會兒 FFT 我們會利用一點小技巧加速到 $\mathcal{O}(n\lg n)$，這裡注意 $P(x)Q(x)$ 的次數最高只能是 $N-1$，否則會得到唯一一個多項式 $H(x)$ 使得 $H(\text{◔}^i)=P(\text{◔}^i)Q(\text{◔}^i)$ 且次數小於 N 的多項式。

為了說明 DFT 的正確性，以下程式碼比較 DFT（on $\mathbb{C}$）跟直接做多項式相乘的差別，感受一下 DFT 的正確性，注意目前兩者都是 $\mathcal{O}(n^2)$。

```
using namespace std;
typedef complex<double> cpx;
#define MAXN 1000
#define PI acos(-1)
#define I cpx(0,1)

cpx D_N[ MAXN ][ MAXN ], D_N_inv[ MAXN ][ MAXN ];
cpx P[ MAXN ],Q[ MAXN ];
cpx pointValueP[ MAXN ], pointValueQ[ MAXN ];
cpx dotProduct[ MAXN ], DFTAns[ MAXN ], directlyMultiplyAns[ MAXN ];

void matrixMultiply(cpx Matrix[ MAXN ][ MAXN ],cpx *Value, cpx *Ans, int
    n){
  for(int i = 0 ; i < n ; ++ i ){
```

```
    Ans[ i ] = 0 ;
    for(int j = 0; j < n ; ++ j ){
      Ans[ i ] += Matrix[ i ][ j ] * Value[ j ];
    }
  }
}
void directlyPolynomialMultiply(cpx *polyP, cpx *polyQ, cpx *Ans, int n){
  for(int i = 0 ; i < n ; ++ i ){
    Ans[ i ] = 0 ;
    for(int j = 0 ; j <= i ; ++ j)
      Ans[ i ] += polyP[ j ] * polyQ[ i - j ];
  }
}
void matrixInitialization(cpx D_N[MAXN][MAXN],cpx D_N_inv[MAXN][MAXN],int
     finalDegree){
  for(int i = 0 ; i < finalDegree ; ++ i){//D_N矩陣與反矩陣
    for(int j = 0; j < finalDegree; ++ j){
      D_N[ i ][ j ] = exp( PI * 2 * ( i * j ) / finalDegree * I );
      D_N_inv[ i ][ j ] = ( cpx ) ( 1. / finalDegree ) * conj( D_N[ i ][
          j ] );
    }
  }
}
void DFT(cpx P[MAXN],cpx Q[MAXN],cpx DFTAns[MAXN],int finalDegree){
  matrixMultiply( D_N, P, pointValueP, finalDegree);
  matrixMultiply( D_N, Q, pointValueQ, finalDegree);
  for(int i = 0; i < finalDegree ; ++ i)
    dotProduct[ i ] = pointValueP[ i ] * pointValueQ[ i ];
  matrixMultiply( D_N_inv, dotProduct, DFTAns, finalDegree);
}
int main(){
  const int PolynomialMaxValue = 50, maxDegree = 500, finalDegree =
      maxDegree * 2;
  for(int i = 0 ; i < maxDegree ; ++ i){//隨機給予兩個多項式初始值
    P[ i ] = rand() % PolynomialMaxValue,
    Q[ i ] = rand() % PolynomialMaxValue;
  }
  matrixInitialization( D_N , D_N_inv , finalDegree);

  DFT(P,Q,DFTAns,finalDegree);

  directlyPolynomialMultiply( P, Q, directlyMultiplyAns, finalDegree);
  for(int i = 0; i < finalDegree ; ++ i ){
    if( abs( DFTAns[ i ] - directlyMultiplyAns[ i ] ) > 1e-5 )
      puts("test fail");
  }
}
```

請讀者自行練習寫出 $\mathbb{Z}_p^*$ 上的 DFT

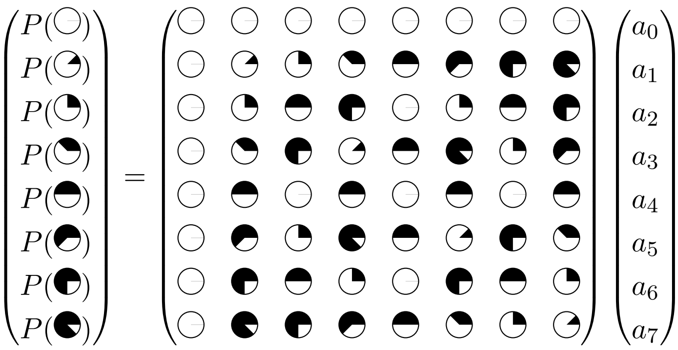
1.3.3 快速傅立葉變換 Fast Fourier Transformation

「快速傅立葉變換」聽起來真是嚇死人了，好像很難的樣子，可以叫他「利用奇怪矩陣的特性所做的加速來算特定幾個點的值之演算法」，講白了就是比較快的 DFT，利用 DFT 中，D_N 的特性所做的加速，它的輸入輸出跟 DFT 沒什麼差別，除了：

1. N 一定要是小的質數的乘積，這裡我們指 2 的次方
2. 比較快，DFT 是 $\mathcal{O}(n^2)$，FFT 只要 $\mathcal{O}(n \lg n)$

為了方便說明，我們這裡一樣舉 $N = 8$ 為例

令 $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$ 要算出



需要這些資訊：
令

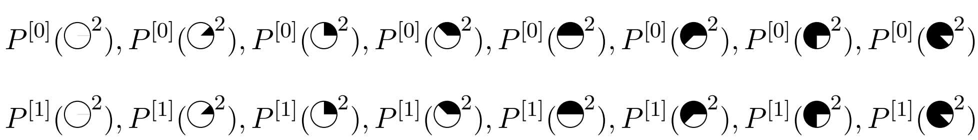
$$P^{[0]}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$$

$$P^{[1]}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$$

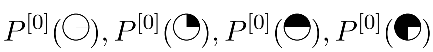
則

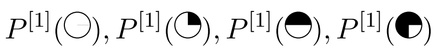
$$P(x) = P^{[0]}(x^2) + x \cdot P^{[1]}(x^2)$$

看起來我們總共要算出

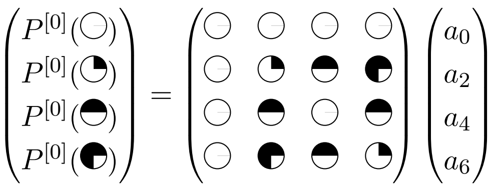


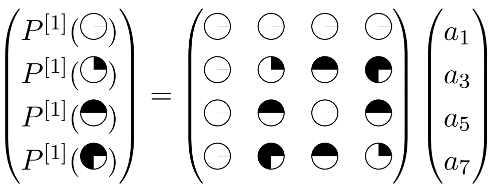
把平方乘進去發現需要的只有





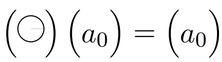
意思是說我們只要算出





便可以在 $\mathcal{O}(n)$ 時間內求出原來的式子。

瞧！發生了什麼事？
只要把一個 FFT 問題分解成兩個小的 FFT 問題，需要的運算量一瞬間少了一半呢！
當問題變得最小時，要解的矩陣像這樣



寫成分析式是這樣：

$$T(N) = 2T(\frac{N}{2}) + \Theta(N)$$

哇～！原來 FFT 就跟 MergeSort 一樣簡單呢！複雜度分析出來是 $\Theta(N \lg N)$，這真的是太棒了！！！乘上 D_N^{-1} 時，注意到 $D_N^{-1} = \frac{1}{N}\overline{D_N}$，因此改一個小小的根，然後結果再乘上 $\frac{1}{N}$ 就好了

下列程式碼比較使用 FFT 算跟剛剛三個步驟程式碼看起來像這樣：

```
using namespace std;
typedef complex<double> cpx;
#define MAXN (1 << 15)
#define PI acos(-1)
#define I cpx(0, 1)
#define EPS 1e-4

cpx dotProduct[MAXN], DFTAns[MAXN];
vector<cpx> P(MAXN), Q(MAXN), pointValueP(MAXN), pointValueQ(MAXN);
vector<cpx> FFTAns(MAXN), directlyMultiplyAns(MAXN);

vector<cpx> directlyPolynomialMultiply(
  const vector<cpx> &PolyP,
```

```
  const vector<cpx> &PolyQ,
  int n
) {
  vector<cpx> ret( n);
  for (int i = 0; i < n; ++i) {
    ret[i] = 0;
    for(int j = 0; j <= i; ++j)
      ret[i] += PolyP[j] * PolyQ[i - j];
  }
  return ret;
}

vector<cpx> FFT(const vector<cpx> &P, int n, cpx root) {
  if (n == 1) return P
  const int half_n = n/2;
  vector<cpx> ret(n, 0), oddP(half_n, 0), evenP(half_n, 0);
  for (int i = 0; i < half_n; ++i)
    evenP[i] = P[2 * i],
    oddP[i]  = P[2 * i + 1];
  evenP = FFT(evenP, half_n, root * root);
  oddP = FFT(oddP, half_n, root * root);
  cpx base = 1;
  for (int i = 0; i < n; ++i) {
    ret[i] = evenP[i % half_n] + oddP[i % half_n] * base;
    base *= root;
  }
  return ret;
}

int main() {
  const int PolynomialMaxValue = 50, maxDegree = 512, finalDegree =
      maxDegree * 2;
  // 隨機給予兩個多項式初始值
  for (int i = 0; i < maxDegree; ++i)
    P[i] = rand() % PolynomialMaxValue,
    Q[i] = rand() % PolynomialMaxValue;

  directlyMultiplyAns = directlyPolynomialMultiply(P, Q, finalDegree);

  P = FFT(P , MAXN , exp(2*PI/MAXN*I));
  Q = FFT(Q , MAXN , exp(2*PI/MAXN*I));
  for (int i = 0; i < MAXN; ++i)
    FFTAns[i] = P[i] * Q[i] * (1./MAXN);
  FFTAns = FFT(FFTAns, MAXN, exp(-2*PI/MAXN*I));

  for (int i = 0; i < finalDegree; ++i) {
    if (abs(FFTAns[i] - directlyMultiplyAns[i]) > EPS)
      puts("test fail");
  }
}
```

程式碼 1.10: FFT: 快速乘 DFT 矩陣的實現

將常數變小	經典問題
1. 排序問題有個技巧是：當問題夠小的時候，直接使用 $\mathcal{O}(n^2)$ 的插入排序，因為問題夠小時（一般來說陣列長度小於 30），插入排序的常數表現非常優。試試看在矩陣夠小時，直接乘矩陣會稍微快一些。 2. 將 FFT 寫成迭代的，請讀者回去查詢使用 bitReverse 的迭代 FFT。	

Golf Bot	UVa 12879
非負整數集 $G = \{0, a_1, a_2, a_3..., a_n\}, a_n \leq 2 \times 10^5$，可以從 G 選兩個元素（可重複挑選同一種元素），請問有沒有辦法選出兩個元素使得它們的和為 m？對於各種指定的 m 請輸出有沒有辦法達成	

這題可以用`std::bitset`寫 $\mathcal{O}(n^2)$ 的算法，但這題若改成。

Golf Bot 改	UVa 12879 改
非負整數集 $G = \{0, a_1, a_2, a_3..., a_n\}, a_n \leq 2 \times 10^5$，可以從 G 選兩個元素（可重複挑選同一種元素），請問有沒有辦法選出兩個元素使得它們的和為 m？對於各種指定的 m 請輸出有幾種辦法達成	

就老實地使用 FFT 吧！

下面有一題可以使用 FFT 以及快速冪就可以算出來的題目

Thief in a Shop	CF 632E
一個小偷要從商店偷剛好 k 個物品，有 N 種物品，每種物品都有無限多個，第 i 種物品價值 a_i 元，問他離開商店時，背包裡的物品價值可能是多少？	