# Advanced Competitive Programming

國立成功大學ACM-ICPC程式競賽培訓隊
nckuacm@imslab.org

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

# Number Theory

# Number Theory

- Prime Generation
- Integer Factorization

# Number Theory

- Prime Generation
- Integer Factorization

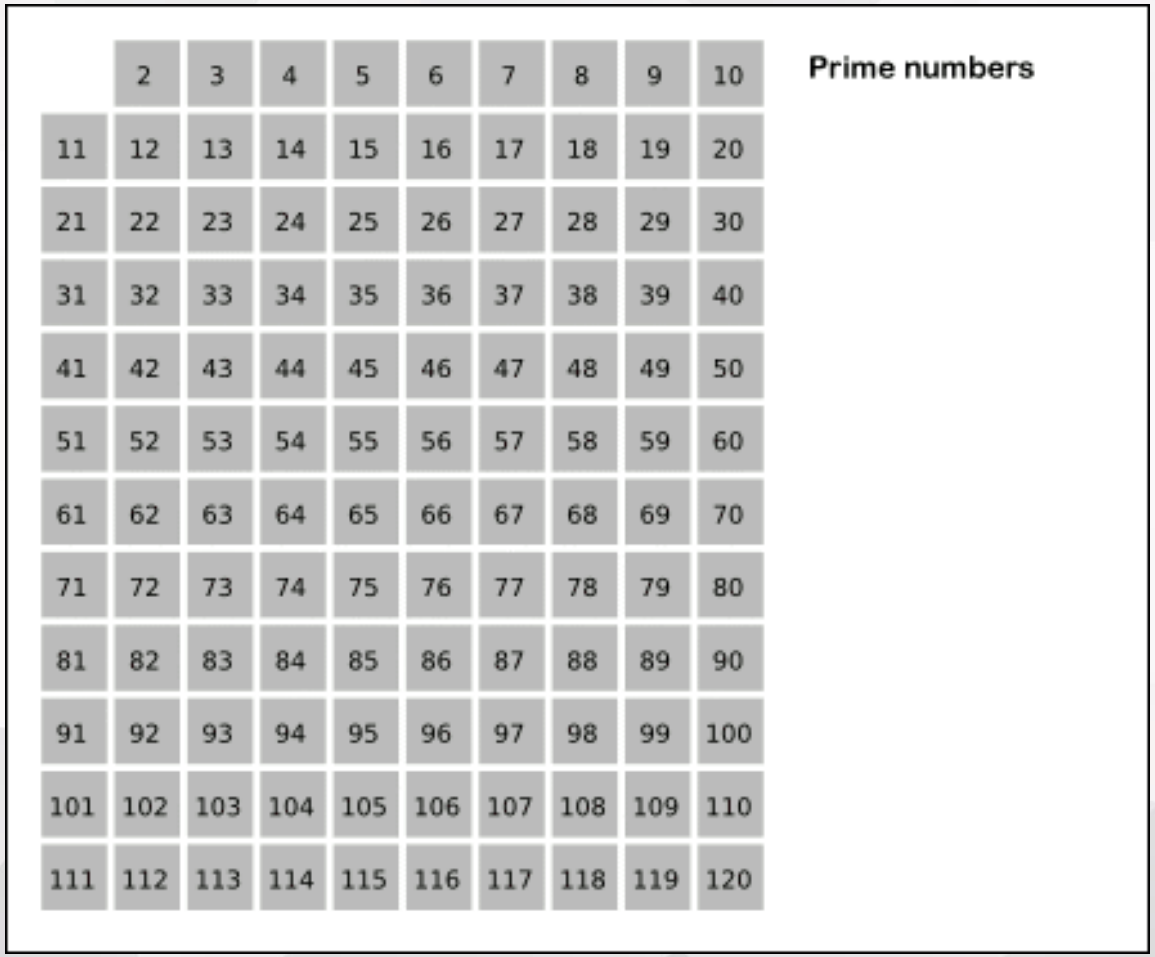
# 埃拉托斯特尼篩法 Sieve of Eratosthenes

簡稱「篩法」

這是一個製作質數表的演算法。

時間複雜度 O(NloglogN)

# 埃拉托斯特尼篩法 Sieve of Eratosthenes




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |

# 埃拉托斯特尼篩法 Sieve of Eratosthenes

列出所有正整數。
從 2 開始，刪掉 2 的倍數。
找下一個未被刪掉的數字，找到 3 ，刪掉 3 的倍數。
找下一個未被刪掉的數字，找到 5 ，刪掉 5 的倍數。

如此不斷下去，就能刪掉所有合數，找到所有質數。

# 埃拉托斯特尼篩法 Sieve of Eratosthenes

```
bool sieve[20000000];
void eratosthenes() {
  sieve[0] = sieve[1] = true; // 0 和 1 不是質數
  for (int i = 2; i < 20000000; i++)
    if (!sieve[i]) // 找下一個未被刪掉的數字
      for (int j = i + i; j < 20000000; j += i)
        sieve[j] = true; // 刪掉質數 i 的倍數
}
```

# 埃拉托斯特尼篩法 Sieve of Eratosthenes

欲刪掉質數 i 的倍數之時，早已刪掉其 2 倍到 i-1 倍了，所以可以直接從 i 倍開始。

一個合數 n ，必定有一個小於等於 sqrt(n) 的質因數。

所有小於等於 sqrt(n) 的質數，刪掉這些質數的倍數，就能刪掉所有合數了，剩下沒刪掉的都是質數

# 埃拉托斯特尼篩法 Sieve of Eratosthenes

```
bool sieve[20000000];
void eratosthenes() {
  sieve[0] = sieve[1] = true;
  for (int i = 2; i * i < 20000000; i++)
                    // 以平方代替根號計算，以避免小數造成誤差
    if (!sieve[i])
      for (int j = i * i; j < 20000000; j += i)
        sieve[j] = true;
}
```

# 線性時間篩法 Linear Sieve Algorithm

一邊製作質數表，一邊刪掉每個數的質數倍，

如此每個合數就只讀取一次，時間複雜度達到 O(N) 。

# 線性時間篩法 Linear Sieve Algorithm

```
const int N = 20000000;
bool sieve[N];
void linear_sieve() {
  vector<int> prime;
  for (int i = 2; i < N; i++) {
    if (!sieve[i]) prime.push_back(i);
    for (int j = 0; i * prime[j] < N; j++) {
      sieve[i * prime[j]] = true;
      if (i % prime[j] == 0) break;
    }
  }
}
```

# Questions?

# 練習

UVa OJ 406

UVa OJ 543

UVa OJ 10140

UVa OJ 10311

# Number Theory

- Prime Generation
- Integer Factorization

# 質因數分解

把一個正整數分解成質因數的連乘積。

$$n = 2^{n_1} \times 3^{n_2} \times 5^{n_3} \times 7^{n_4} \times 11^{n_5} \times \ldots$$

# 質因數分解

把所有可能的因數拿來試除。

```
void trial_division(int n) {
  for(int d = 2; d <= n; ++d)
    while(n % d == 0) {
      n /= d;
      cout << d; // 質因數
    }
}
```

# 質因數分解

跟篩質數時一樣檢查小於等於sqrt(n)的因數就好了

```
void trial_division(int n) {
  for(int d = 2; d * d <= n; ++d)
    while(n % d == 0) {
      n /= d;
      cout << d; // 質因數
    }
  if(n > 1) cout << n; // n是質數
}
```

# 質因數分解

質因數必定是質數

所以只要建好質數表
然後試除小於sqrt(n)的質數就好了

# Questions?

# 練習

UVa OJ 583

UVa OJ 10392

UVa OJ 10622

UVa OJ 10791

UVa OJ 10879

# Calculation

# Calculation

對於**大數字**的運算，普通的做法不夠快，

因此接下來將介紹快速的**乘法**及**冪**運算

# Calculation

- Gauss's complex multiplication algorithm
- Karatsuba algorithm
- Fast exponentiation

# Calculation

- Gauss's complex multiplication algorithm
- Karatsuba algorithm
- Fast exponentiation

# 複數乘法

對於 a + bi, c + di

# 複數乘法

對於 a + bi, c + di

相乘得 $(ac - bd) \cdot (bc + ad)i$

# 複數乘法

對於 a + bi, c + di

相乘得 (ac – bd) · (bc + ad)i

一般需要計算 ac, bd, bc, ad 共四次乘法
才能計算出兩數相乘

# 複數乘法

$(ac - bd) \cdot (bc + ad)i$

對於 $(ac - bd) = ac + bc - bc - bd$

# 複數乘法

$(ac - bd) \cdot (bc + ad)i$

對於 $(ac - bd) = \underline{ac + bc} - \underline{bc - bd}$

令

$$k_1 = ac + bc = c(a+b)$$
$$k_2 = bc + bd = b(c+d)$$

# 複數乘法

$(ac - bd) \cdot (bc + ad)i$

對於 $(bc + ad) = ac + bc + ad - ac$

# 複數乘法

$(ac - bd) \cdot (bc + ad)i$

對於 $(bc + ad) = \underline{ac + bc} + \underline{ad - ac}$

令

$$k_1 = ac + bc = c(a+b)$$
$$k_3 = ad - ac = a(d-c)$$

# 複數乘法

$(ac - bd) \cdot (bc + ad)i$

$= (k_1 - k_2) \cdot (k_1 + k_3)i$

$$k_1 = ac + bc = c(a+b)$$
$$k_2 = bc + bd = b(c+d)$$
$$k_3 = ad - ac = a(d-c)$$

# 複數乘法

$(ac - bd) \cdot (bc + ad)i$

$= (k_1 - k_2) \cdot (k_1 + k_3)i$

總共只需要三次乘法

似乎變快了一點

# Questions?

# Calculation

- Gauss's complex multiplication algorithm
- Karatsuba algorithm
- Fast exponentiation

# Karatsuba algorithm

對於剛剛的複數乘法

似乎對於整數 x, y 相乘有些啟示

# Karatsuba algorithm

對於剛剛的複數乘法

似乎對於整數 x, y 相乘有些啟示

也就是令

$$x = am + b$$
$$y = cm + d$$

# Karatsuba algorithm

$$x = am + b$$
$$y = cm + d$$

$$xy = acm^2 + (ad + bc)m + bd$$

# Karatsuba algorithm

$$xy = acm^2 + \underline{(ad + bc)}m + bd$$

其中

$$(ad + bc) = ad + ac + bc + bd - bd - ac$$

$$= (a + b)(c + d) - bd - ac$$

# Karatsuba algorithm

$$xy = acm^2 + \underline{(ad + bc)}m + bd$$

$$(ad + bc) = ad + ac + bc + bd - bd - ac$$
$$= (a + b)(c + d) - bd - ac$$

$z_1 = ac$

$z_2 = bd$

$z_3 = (a + b)(c + d)$

# Karatsuba algorithm

$$xy = acm^2 + \underline{(ad + bc)}m + bd$$

也就是說

$$xy = z_1m^2 + (z_3-z_1-z_2)m + z_2$$

$z_1 = ac$

$z_2 = bd$

$z_3 = (a + b)(c + d)$

# Karatsuba algorithm

$$xy = acm^2 + \underline{(ad + bc)}m + bd$$

也就是說

$$xy = z_1m^2 + (z_3-z_1-z_2)m + z_2$$

$z_1 = ac$

$z_2 = bd$

$z_3 = (a + b)(c + d)$

# Karatsuba algorithm

$$xy = z_1 m^2 + (z_3 - z_1 - z_2)m + z_2$$

假設數字長度為 n

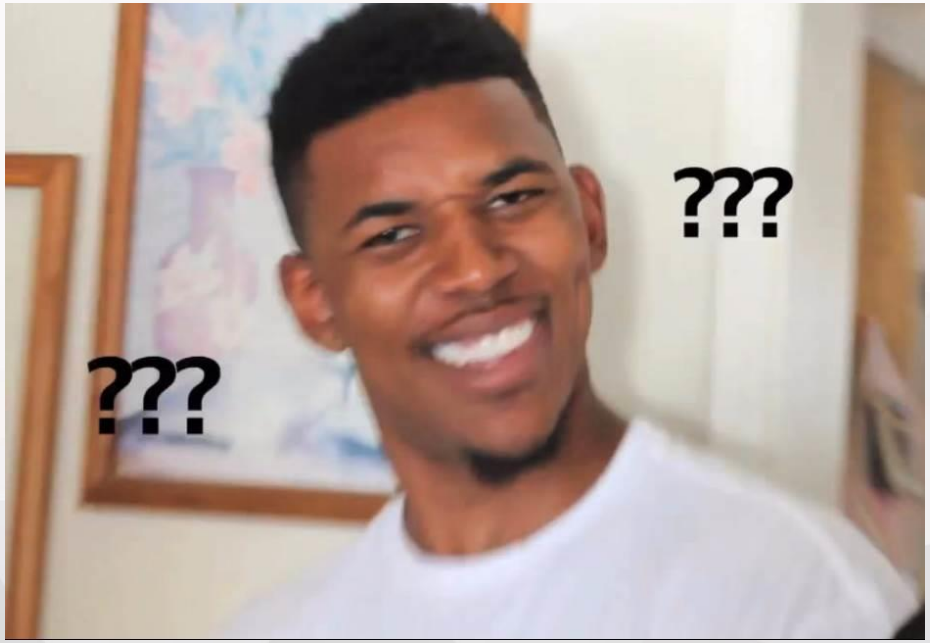
總共只需要三次乘法

相較於直式乘法複雜度 $O(n^2)$

這個算法有複雜度 $3 \cdot O(n^2)$

# Karatsuba algorithm

到底在幹三小?



# 分治法

對於上述演算法，
凡是遇到乘法運算，都使用同樣的演算法

# 分治法

對於上述演算法，
凡是遇到乘法運算，都使用同樣的演算法

並且對於數字的分割，總是分成均等的**兩半**

例如 6789 分成 67 和 89
$6789 = 67 \cdot 100 + 89$

# 分治法

```
int k(int x, int y) {
  if(x < 10 || y < 10) return x*y;

  int len = min(log10(x), log10(y));
  int m = pow(10, len/2 + 1);

  auto [a, b] = div(x, m);
  auto [c, d] = div(y, m);

  int z1 = k(a, c), z2 = k(b, d), z3 = k(a+b, c+d);
  return z1*m*m + (z3-z1-z2)*m + z2;
}
```

# 分治法

時間成本 T(n)

$T(n) = 3 \cdot T(n/2) + c_n$

$T(1) = 1 + c_1$

複雜度為 $O(3^{\lg n}) = O(n^{\lg 3})$

# Questions?

# Calculation

- Gauss's complex multiplication algorithm
- Karatsuba algorithm
- Fast exponentiation

# Exponentiating by Squaring

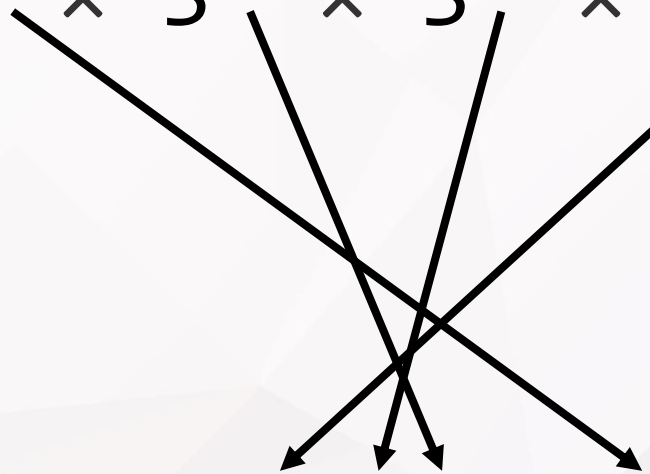
快速冪以及矩陣快速冪

# 如何計算 $3^{987654321} \% 1000007$

- O(n)：反正就把 3 一直乘
- O(lg n)：快速冪

# 快速冪

- 觀察 $3^n \times 3^n = 3^{2n}$
- 可以分解$3^{987654321} = 3^1 \times 3^{16} \times 3^{32} \times 3^{128} \times \cdots\cdots$



987654321

$= 111010110111100110100010110001_{(2)}$// 抱歉很亂

$= 1 + 16 + 32 + 128 + \ldots$

# 快速冪

- 只要 n 是 2 的冪次 $3^n$ 就能很快求出來
- $3^n = 3^{n/2} \times 3^{n/2}$

# 快速冪

```
int x = 1, a = 3;
while (n) {
  if (n&1) x *= a; // 二進位尾數是 1
  x %= 1000007;
  a *= a;
  a %= 1000007;
  n >>= 1;
} // x 就是答案
```

# 矩陣快速冪

- 矩陣也有類似性質
- 假設現在有個方陣 $A$，$A^n \times A^n = A^{2n}$
- 對於費氏數列：

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_n \end{bmatrix}$$

# 練習 Zero Judge b525

- b525: 先別管這個了，你聽過turtlebee嗎？

# Questions?

# Floating-Point Precision

浮點數誤差以及競程處理技巧

# 形成原因：IEEE 754 的浮點數的儲存

- 用 0 跟 1 表示浮點數
- 表示方式：$1.1101010_{(2)} \times 2^{4}{}_{(10)}$
- 優點：在精度內可以表達好，通常精度夠用
- 缺點：超出精度就無法表示
- 直接 WA

# 舉例

- $0.69_{(\text{double})} \times 10 \neq 6.9_{(\text{double})}$
- 結果可能是 6.89999……
- 在 print 的時候不容易出錯，但在比較大小或判斷相等時常會出問題

# 解決方法

- 假設今天有個閾值是 6.9，6.899999 這樣的表示會有問題
- 解決方案一：四捨五入
- 解決方案二：乘上 1.000…001(數量根據精度調整)
  -> 較推薦

# 解決方法(比較大小)

```
if (0.69 * 10 * 1.000…1 >= 6.9) {
    do something
}
```

# 解決方法(比較相等)

```
if (abs((0.69*10) - 6.9) <= 1e-15) {
    do something
}
```

# AC Get