Algorithm Notes

by Ham Kittichet

May 27, 2025

▶ Table of Contents

บทที่ 1. Divide-and-Conquer **1**

▶ **1.1.** ปัญหา Maximum Subarray 1

▶ **1.2.** อัลกอริทึมการคูณเมทริกซ์ของ Strassen 2

▶ **1.3.** ความสัมพันธ์เวียนเกิด 4

บทที่ 1 | Divide-and-Conquer

▶ 1.1. ปัญหา Maximum Subarray

สมมติเรามีลำดับของจำนวนจริง (array) ชุดหนึ่ง และเราต้องการหาลำดับย่อยที่เรียงติดกันที่มีผลรวมมากที่สุด (จะเรียกว่าเป็น *maximum subarray*) เราอาจจะทำตรง ๆ เลยโดยเช็คทุก ๆ ลำดับย่อยที่เป็นไปได้ซึ่งถ้าลำดับนี้มีจำนวนสมาชิกอยู่ n ตัว จะทำให้ต้องเช็คลำดับย่อยทั้งหมด $\binom{n}{2}$ ชุด จึงต้องใช้เวลา $\Omega(n^2)$

อีกวิธีที่ดีกว่าคือการใช้ recursion โดยเราจะแบ่ง array ที่ได้รับมานี้ออกเป็น 2 subarray (โดยจะเก็บ index ไว้สามตัวคือ *low*, *mid*, และ *high*) ไปเรื่อย ๆ และหา maximum subarray ของ subarray ซ้าย, subarray ขวา, และ maximum subarray ที่ข้ามจุดแบ่ง (crossing subarray) จากนั้นเลือกค่าที่มากที่สุดในสามกรณีนี้

สังเกตว่าเราสามารถหา maximum crossing subarray ของ array A ขนาด n ที่ผ่าน mid ได้โดยการหา maximum subarray ของครึ่งซ้ายรวมกับของครึ่งขวา:

(1.1)

การหา Maximum Crossing Subarray.

```
1  ฟังก์ชัน findMaxCrossingSubarray(A, low, mid, high)
2      leftSum ← −∞
3      sum ← 0
4      สำหรับ i ← mid ลงไปถึง low ให้ทำ
5          sum ← sum + A[i]
6          ถ้า sum > leftSum แล้ว
7              leftSum = sum
8              maxLeft = i
9      ทำคล้ายกันสำหรับฝั่งขวา โดยลูปตั้งแต่ mid + 1 ถึง high
10     รีเทิร์น (maxLeft, maxRight, leftSum + rightSum)
```

ซึ่งใช้เวลา $\Theta(n)$

ดังนั้นก็จะได้อัลกอริทึมในการหา maximum subarray โดยการ *divide-and-conquer*:

(1.2)

อัลกอริทึม Maximum Subarray โดยใช้ Divide-and-Conquer.

```
1  ฟังก์ชัน findMaxSubarray(A, low, high)
2      ถ้า low = high แล้ว
3          รีเทิร์น (low, high, A[low])          ▷ ขั้นฐานเมื่อ subarray มีขนาดเป็น 1
4      mid ← ⌊(low + high)/2⌋
5      (left)_{i=1}^3 ← findMaxSubarray(A, low, mid)
6      (mid)_{i=1}^3 ← findMaxCrossingSubarray(A, low, mid, high)
7      (right)_{i=1}^3 ← findMaxSubarray(A, mid + 1, high)
8      maxSubarray ← เลือก x ∈ {left, mid, right} ที่มีค่าของ x_3 มากที่สุด
9      รีเทิร์น (maxSubarray)_{i=1}^3
```

โดยเราจะเรียก $\texttt{findMaxSubarray}(A, 1, A.length)$ เมื่อต้องการหา maximum subarray ของ A

ถ้ากำหนดให้อัลกอริทึมนี้ทำงานได้ในเวลา $T(n)$ ก็จะได้ความสัมพันธ์

(1.3)

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$$

(เพราะการหา *left* และ *right* เป็นการเรียกฟังก์ชันเดิมนี้ซ้ำ โดยที่ array มีขนาดลดลงครึ่งหนึ่ง ใช้เวลา $T(\lfloor n/2 \rfloor)$, การหา *mid* ใช้เวลา $\Theta(n)$, และที่เหลือทั้งหมดใช้เวลา $\Theta(1)$) โดยในส่วนถัด ๆ ไปเราจะแก้ได้ว่าความสัมพันธ์เวียนเกิดนี้มีคำตอบ $T(n) = \Theta(n \lg n)$ ซึ่งเร็วกว่าการทำตรง ๆ

▶ 1.2. อัลกอริทึมการคูณเมทริกซ์ของ Strassen

▶ การคูณเมทริกซ์จตุรัสโดยใช้ Divide-and-Conquer แบบตรง ๆ

จากที่ได้เห็นว่าการใช้ divide-and-conquer อาจทำให้ได้อัลกอริทึมที่ไวกว่าการทำปกติ เราลองมาพยายามใช้ divide-and-conquer กับการคูณเมทริกซ์จตุรัส โดยแบ่งเมทริกซ์ $n \times n$ เป็น 4 เมทริกซ์ที่มีขนาด $n/2 \times n/2$ (อาจปัดขึ้น-ปัดลงได้เพื่อให้เป็นจำนวนเต็ม แต่จะละไว้ในฐานที่เข้าใจเพราะจริง ๆ แล้วเราสามารถพิสูจน์ได้ว่าการปัดขึ้น-ปัดลงไม่ส่งผลต่อ asymptotic runtime ของอัลกอริทึม) แล้วค่อยนำเมทริกซ์ย่อยมาคูณกัน ดังนี้

(1.4)

อัลกอริทึมการคูณเมทริกซ์จตุรัสโดยใช้ Divide-and-Conquer.

```
1  ฟังก์ชัน recurMatrixMultiply(A, B)
2      n ← A.rows
3      ให้ C เป็นเมทริกซ์ใหม่ที่แทนผลลัพธ์ของ A × B
4      ถ้า n = 1 แล้ว
5          c_11 ← a_11 b_11                  ▷ ขั้นฐานในกรณี n = 1 ให้คูณแบบสเกลาร์ปกติ
6      มิเช่นนั้น
```

(1.4)

```
7        แบ่งเมทริกซ์ A, B, และ C ออกเป็น A_11, A_12, A_21, A_22, ... ที่มีขนาด n/2 × n/2
8        C_11 ← recurMatrixMultiply(A_11, B_11) + recurMatrixMultiply(A_12, B_21)
9        C_12 ← recurMatrixMultiply(A_11, B_12) + recurMatrixMultiply(A_12, B_22)
10       C_21 ← recurMatrixMultiply(A_21, B_11) + recurMatrixMultiply(A_22, B_21)
11       C_22 ← recurMatrixMultiply(A_21, B_11) + recurMatrixMultiply(A_22, B_21)
12   รีเทิร์น C
```

หมายเหตุ: *การคูณทั้งหมดนี้เป็นการคูณจาก index จึงไม่ต้องเสียเวลาในการคัดลอกข้อมูลมาใส่เมทริกซ์ใหม่ทุกการ call*

โดยถ้าอัลกอริทึมนี้ใช้เวลา $T(n)$ ในการคูณเมทริซ์จตุรัสขนาด $n \times n$ เรามาพิจารณาความสัมพันธ์เวียนเกิดของ T โดยเริ่มจากการแบ่งเมทริกซ์ ใช้เวลา $\Theta(1)$ (เพราะเป็นการคูณโดยแบ่งจาก index) และถัดมาเราจะต้องเรียกฟังก์ชันนี้ซ้อนไปอีก 8 รอบ ใช้เวลา $8T(n/2)$ สุดท้าย ในการนำเมทริกซ์ย่อยที่คูณกันแล้วมาบวกกัน ใช้เวลา $\Theta(n^2)$ ดังนั้น

(1.5)

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

ซึ่งเมื่อใช้ master theorm (ในส่วนถัดไป) จะได้ว่า $T(n) = \Theta(n^3)$ ซึ่งไม่ต่างอะไรกับการเขียนแบบคูณตรง ๆ จึงไม่ได้มีประโยชน์อะไรมากนัก

▶ การคูณเมทริกซ์จตุรัสด้วยอัลกอริทึมของ Strassen

แต่ Volker Strassen ได้ค้นพบวิธีทั่วไปที่เร็วกว่าแบบทำตรง ๆ ที่มีไอเดียมาจาก divide-and-conquer เช่นกัน โดยสังเกตว่าจาก (1.5) การบวกเมทริกซ์ไม่ได้ส่งผลกับความสัมพันธ์เวียนเกิดเท่าไหร่ เพราะไม่ว่าจะบวกกันกี่ครั้ง ถ้าจำนวนครั้งในการบวกคงที่ สุดท้ายแล้วพจน์หลังก็จะเป็น $\Theta(n^2)$ อยู่ดี แต่ในทางกลับกันจำนวนครั้งในการคูณเมทริกซ์ส่งผลโดยตรงกับสัมประสิทธ์ด้านหน้า $T(n/2)$ ดังนั้นถ้าเราสามารถลดจำนวนครั้งในการคูณเมทริกซ์โดยแลกกับการบวกเมทริกซ์เพิ่มอีกนิด ตามเซนส์แล้วน่าจะทำให้ time complexity สุดท้ายลดลง (เหตุผลจริง ๆ คือจาก master theorem แล้ว $n^{\log_2 8}/n^2 = n^1$ แปลว่ายังสามารถลด 8 ลงไปได้อีกในขณะที่ยังไม่ทำให้พจน์ $\Theta(n^2)$ โตเร็วเกินไป)

โดยอัลกอริทึมของ Strassen ลดจำนวนการคูณเมทริกซ์ย่อยจาก 8 ครั้งเหลือเพียง 7 ครั้ง แลกกับการบวกและลบเมทริกซ์เพิ่มขึ้น ดังนี้:

(1.6)

อัลกอริทึมการคูณเมทริกซ์จตุรัสของ Strassen.

```
1  ฟังก์ชัน strassenMatrixMultiply(A, B)
2      n ← A.rows
3      ถ้า n = 1 แล้ว
4          c_11 ← a_11 b_11
5      มิเช่นนั้น
6          แบ่งเมทริกซ์ A, B, และ C ออกเป็น A_11, A_12, A_21, A_22, ... ที่มีขนาด n/2 × n/2
7          P_1 ← strassenMatrixMultiply(A_11 + A_22, B_11 + B_22)
```

(1.6)

```
8    P_2 ← strassenMatrixMultiply(A_21 + A_22, B_11)
9    P_3 ← strassenMatrixMultiply(A_11, B_12 − B_22)
10   P_4 ← strassenMatrixMultiply(A_22, B_21 − B_11)
11   P_5 ← strassenMatrixMultiply(A_11 + A_12, B_22)
12   P_6 ← strassenMatrixMultiply(A_21 − A_11, B_11 + B_12)
13   P_7 ← strassenMatrixMultiply(A_12 − A_22, B_21 + B_22)
14   C_11 ← P_1 + P_4 − P_5 + P_7
15   C_12 ← P_3 + P_5
16   C_21 ← P_2 + P_4
17   C_22 ← P_1 − P_2 + P_3 + P_6
18 รีเทิร์น C
```

ซึ่งจะได้ความสัมพันธ์เวียนเกิดของเวลาในการทำงานคือ

(1.7)

$$T(n) = 7T(n/2) + \Theta(n^2)$$

ซึ่งเมื่อใช้ master theorem จะได้ว่า $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$ เร็วกว่าการคูณแบบปกติที่ใช้เวลา $\Theta(n^3)$

การคูณและนำมาประกอบกันแบบนี้ดูเหมือน *black magic* แต่ไอเดียหลัก ๆ คือเมื่อลองพิจารณาจากความสัมพันธ์เวียนเกิด เราสามารถสังเกตได้ว่าการบวก “คุ้ม” กว่าการคูณมากในแง่ของเวลาในการทำงาน ดังนั้นหากสามารถลดจำนวนการคูณลงได้ แม้จะต้องเพิ่มจำนวนการบวกขึ้น ก็จะทำให้ time complexity โดยรวมลดลงอย่างมีนัยสำคัญ

▶ 1.3. ความสัมพันธ์เวียนเกิด