Algorithm Notes

by Ham Kittichet

May 29, 2025

▶ Table of Contents

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  Function findMaxCrossingSubarray(A, low, mid, high)
2      leftSum ← −∞
3      sum ← 0
4      for i ← mid downto low do
5          sum ← sum + A[i]
6          if sum > leftSum then
7              leftSum = sum
8              maxLeft = i
9      ทำคล้ายกันสำหรับฝั่งขวา โดยลูปตั้งแต่ mid + 1 ถึง high
10     return (maxLeft, maxRight, leftSum + rightSum)
```

ซึ่งใช้เวลา $\Theta(n)$

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

(1.2)

อัลกอริทึม Maximum Subarray โดยใช้ Divide-and-Conquer.

```
1 Function findMaxSubarray(A, low, high)
2     if low = high then
3         return (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     return (maxSubarray)_{i=1}^3
```

โดยเราจะเรียก `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 Function recurMatrixMultiply(A, B)
2     n ← A.rows
3     ให้ C เป็นเมทริกซ์ใหม่ที่แทนผลลัพธ์ของ A × B
4     if n = 1 then
5         c_11 ← a_11 b_11                  ▷ ขั้นฐานในกรณี n = 1 ให้คูณแบบสเกลาร์ปกติ
6     else
```

(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   return 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 แต่ในทางกลับกันจำนวนครั้งในการคูณเมทริกซ์ส่งผลโดยตรงกับสัมประสิทธิ์ด้านหน้า $T(n/2)$ 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  Function strassenMatrixMultiply(A, B)
2      n ← A.rows
3      if n = 1 then
4          c_11 ← a_11 b_11
5      else
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)
8          P_2 ← strassenMatrixMultiply(A_21 + A_22,  B_11)
```

(1.6)

```
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   return C
```

ซึ่งจะได้ความสัมพันธ์เวียนเกิดของเวลาในการทำงานคือ

(1.7)
$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

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

▶ การพิสูจน์ความสัมพันธ์เวียนเกิด

เราสามารถพิสูจน์คำตอบ (แบบ asymptotic) ของความสัมพันธ์เวียนเกิดได้โดยใช้การอุปนัยอย่างเข้ม (strong induction) ตัวอย่างเช่น

(1.8) **ตัวอย่าง.** จงพิสูจน์ว่าความสัมพันธ์เวียนเกิด $T(n) = 2\,T(n/2) + n$ มีคำตอบคือ $T(n) = \Theta(n \lg n)$

พิสูจน์. ก่อนอื่นจะพิสูจน์ว่า $T(n) = O(n \lg n)$ โดยจะแสดงว่ามี $n_0 \in \mathbb{N}$ และ $c \in \mathbb{R}^+$ ซึ่งทำให้ $T(n) \leq cn \lg n$ ทุก ๆ จำนวนเต็ม $n > n_0$

ขั้นอุปนัย: สมมติ $T(t) \leq ct \lg t$ ทุก ๆ จำนวนเต็ม $n_0 < t < n$ จะได้ว่า

$$T(n) = 2\,T(n/2) + n \leq cn \lg n - \frac{c}{2}n + n$$

ดังนั้นจะเห็นว่าถ้าเลือก $c \geq 2$ ก็จะทำให้สามารถพิสูจน์ได้ว่าขั้นอุปนัยเป็นจริง

ขั้นฐาน: สังเกตว่าถ้าเราให้ขั้นฐานเป็น $n = 1$ จะได้ว่า

$$T(1) \leq c \lg 1 = 0$$

ซึ่งไม่เป็นจริงอยู่แล้ว เราจึงลองย้ายขั้นฐานมาเป็น $n = 2$ และ 3 (เพราะว่าสำหรับความสัมพันธ์เวียนเกิดที่เป็นจำนวนเต็ม ถ้า T ติดเป็น floor function หรือ ceiling function แทนจะทำให้ทุก ๆ ค่า $T(n)$ เขียนได้อยู่ในรูปของ $T(2)$ และ $T(3)$ ได้เสมอ ไม่จำเป็นต้องติด $T(1)$ ทำให้ขั้นอุปนัยไม่เจ๊ง) ก็จะได้

$$T(2) \leq 2c \lg 2 \qquad T(3) \leq 3c \lg 3$$

ดังนั้นถ้าเราเลือก c ที่มีค่ามากพอจะได้ขั้นฐานและขั้นอุปนัยเป็นจริง

โดยหลักอุปนัยเชิงคริตศาสตร์อย่างเข้ม จึงได้ว่า $T(n) = O(n \lg n)$ (การเขียนพิสูจน์จริง ๆ จะต้องเริ่มจากหาค่า c มาเลย แต่การเขียนแบบนี้จะเห็นชัดกว่าว่าค่า c นี้มาจากไหน)

ในทำนองเดียวกันเราจะสามารถแสดงได้ว่า $T(n) = \Omega(n \lg n)$ และเนื่องจาก $T(n) = O(n \lg n)$ และ $T(n) = \Omega(n \lg n)$ ก็จะได้ว่า $T(n) = \Theta(n \lg n)$ ตามต้องการ □

แต่กสังเกตว่าการพิสูจน์คำตอบแบบนี้เราจะต้อง "เดา" คำตอบของความสัมพันธ์เวียนเกิดให้ได้ก่อน โดยวิธีในการเดาคำตอบที่ดีวิธีหนึ่งคือการพิจารณา *recursion tree* (หรือบางครั้งถ้าเขียนออกมาดีพอ อาจนับว่าเป็นการพิสูจน์ได้เลย) ยกตัวอย่างเช่น

(1.9)

ตัวอย่าง. จงหา asymptotic upper bound ของความสัมพันธ์เวียนเกิด $T(n) = 4\,T(n/2) + n^2 \lg n$

วิธีทำ. พิจารณา recursion tree ของ $T(n)$ (โดยสมมติว่า n เป็นกำลังของ 2) จะได้ว่าเราสามารถ "แตกกิ่ง" ลงมาจนถึง $T(1)$ ได้ $\lg n$ ครั้ง ในชั้นที่ i จะมีจำนวนโหนดทั้งหมด 4^i โหนดดังนั้นแต่ละครั้งที่แตกกิ่งออกมาจากชั้นที่ i ก็จะเกิดพจน์ $\left(n/2^i\right)^2 \lg\left(n/2^i\right) = \left(n^2/4^i\right)(\lg n - i)$ มาทั้งหมด 4^i พจน์ และในชั้นสุดท้ายจะมี $T(1)$ ทั้งหมด $4^{\lg n} = n^2$ พจน์ ดังนั้นจะได้

$$\begin{aligned}
T(n) &= n^2\,T(1) + \sum_{i=0}^{\lg n - 1} \left(\cancel{4^i}\right)\left(\frac{n^2}{\cancel{4^i}}\right)(\lg n - i) \\
&= n^2\,T(1) + n^2 \lg^2 n - n^2 \sum_{i=0}^{\lg n - 1} i \\
&= n^2\,T(1) + n^2 \lg^2 n - \frac{n^2}{2}(\lg^2 n - \lg n) \\
&= n^2\,T(1) + \frac{1}{2}n^2 \lg^2 n + \frac{1}{2}n^2 \lg n
\end{aligned}$$

ดังนั้นจะเห็นได้ว่า $T(n) = O(n^2 \lg n)$ โดยเราสามารถพิสูจน์ได้อย่างเป็นทางการคล้าย ๆ กับข้อที่แล้วว่าเป็นจริง □

▶ Master Theorem

(1.10)

Master Theorem.

สำหรับความสัมพันธ์เวียนเกิด $T(n) = a\,T(n/b) + f(n)$ (โดยที่ n/b อาจเป็นปัดขึ้นหรือปัดลง) จะได้ว่า

(i) ถ้า $f(n) = O\big(n^{\log_b a - \varepsilon}\big)$ สำหรับบาง $\varepsilon > 0$ (กล่าวคือ $f(n)$ โตช้ากว่า $n^{\log_b a}$ *ในเชิงพหุนาม*) แล้ว $T(n) = \Theta\big(n^{\log_b a}\big)$

(ii) ถ้า $f(n) = \Theta\big(n^{\log_b a}\big)$ แล้ว $T(n) = \Theta(f(n)\lg n) = \Theta\big(n^{\log_b a}\cdot \lg n\big)$

(iii) ถ้า $f(n) = \Omega\big(n^{\log_b a+\varepsilon}\big)$ สำหรับบาง $\varepsilon > 0$ ($f(n)$ โตเร็วกว่า $n^{\log_b a}$ ในเชิงพหุนาม) และ $a\,f(n/b) \le c\,f(n)$ สำหรับบางจำนวนจริง $c < 1$ เมื่อ n มีค่ามากพอ แล้ว $T(n) = \Theta(f(n))$

พิสูจน์. (แบบคร่าว ๆ) จะพิสูจน์แค่ในกรณี $n = b^k$ เพื่อให้ได้ intuition

พิจารณา recursion tree ของความสัมพันธ์เวียนเกิดนี้ จะได้ว่าเราสามารถแต่งกิ่งลงไปได้ $\log_b n$ ครั้ง โดยในแต่ละชั้นที่ i จะมีจำนวนโหนด a^i โหนด และแต่ละโหนดจะทำให้เกิดพจน์ $f(n/b^i)$ ดังนั้นผลรวมทั้งหมดคือ

$$T(n) = a^{\log_b n}\,T(1) + \sum_{i=0}^{\log_b n - 1} a^i f\Big(\frac{n}{b^i}\Big) \equiv \Theta\Big(n^{\log_b a}\Big) + g(n) \qquad (\odot)$$

กรณี (i): สมมติ $f(n) = O\big(n^{\log_b a-\varepsilon}\big)$ จะได้ว่า

$$g(n) = O\left(\sum_{i=0}^{\log_b n-1} a^i\Big(\frac{n}{b^i}\Big)^{\log_b a-\varepsilon}\right) = O\left(n^{\log_b a-\varepsilon}\sum_{i=0}^{\log_b n-1} \cancel{a^i}\Big(\frac{b^\varepsilon}{\cancel{a^i}}\Big)\right) = O\Big(n^{\log_b a}\Big)$$

เมื่อนำไปแทนกลับใน $(\odot)$ จะได้ว่า $T(n) = \Theta\big(n^{\log_b a}\big)$

กรณี (ii): สมมติ $f(n) = \Theta\big(n^{\log_b a}\big)$ ก็จะได้ว่า

$$g(n) = \Theta\left(\sum_{i=0}^{\log_b a-1} a^i\Big(\frac{n}{b^i}\Big)^{\log_b a}\right) = \Theta\left(\sum_{i=0}^{\log_b a-1} n^{\log_b a}\right) = \Theta\Big(n^{\log_b a}\lg n\Big)$$

แทนใน $(\odot)$ จะได้ $T(n) = \Theta\big(n^{\log_b a}\lg n\big)$

กรณีที่ (iii): สมมติ $a\,f(n/b) \le c\,f(n)$ สำหรับจำนวนจริง $c < 1$ และ $f(n) = \Omega\big(n^{\log_b a-\varepsilon}\big)$ จากเงื่อนไขแรกเราจะได้ว่า $a^i\,f(n/b^i) \le c^i\,f(n)$ ทุกค่า i ตราบใดที่ n มีค่ามากพอ ดังนั้น

$$g(n) = \sum_{i=0}^{\log_b a-1} a^i f\Big(\frac{n}{b^i}\Big) \le \sum_{i=0}^{\log_b a-1} c^i f(n) \le \frac{1}{1-c} f(n)$$

ก็จะได้ $g(n) = O(f(n))$ (เรื่องที่อสมการนี้จริงก็ต่อเมื่อ n มีค่ามากพอนั้นเราสามารถละได้เพราะสุดท้ายแล้วในการ bound ค่า จะเกิดความต่างจากค่าที่ถูก bound แค่เพียง constant หนึ่ง ซึ่งไม่มีผลผลใน asymptotic bound) รวม

กับการที่ $g(n) = \Omega(f(n))$ (เพราะ $g(n)$ มีพจน์ $f(n)$ อย่างน้อยหนึ่งพจน์แน่ ๆ และพจน์อื่น ๆ ไม่เป็นลบ) จะได้ว่า $g(n) = \Theta(f(n))$ หรือเมื่อแทนใน $(\odot)$ จะได้ $T(n) = \Theta(f(n))$

หมายเหตุ: *จริง ๆ แล้วจะเห็นได้ว่าในกรณี (iii) เราไม่จำเป็นจะต้องมีเงื่อนไข* $f(n) = \Omega\big(n^{\log_b a+\varepsilon}\big)$ *(แต่สามารถพิสูจน์ได้ว่าเงื่อนไข regularity* $af(n/b) \leq cf(n)$ *imply ว่าเงื่อนไขแรกต้องจริงอยู่ดี)*

จากทั้งสามกรณี จะได้ว่า master theorem เป็นจริง (เมื่อ $n = b^k$) ตามต้องการ □

บทที่ 2 | อัลกอริทึมแบบสุ่ม