Dynamic Programming (DP) ต่างจาก Divide & conquer ตรงที่ DP ลดการคำนวณคำตอบ (คำนวณ)
→ จะมีปัญหาขนาดใหญ่ แล้วแบ่งออกเป็น substructure คือ แบ่งเป็นปัญหาย่อยๆ แล้วก็ใช้การทำ recursion + reuse (memorization)
ข้อดี ประหยัดการคำนวณ , เร็ว (n)
ข้อเสีย Save ไว้หมดทำให้กิน Memory

Divide & conquer ข้อเสียคือเสียเวลาหาคำตอบซ้ำๆ ไม่คำนวณ

บทที่ 8 การโปรแกรมเชิงพลวัตร

8.1 การโปรแกรมเชิงพลวัตร

ข้อเสียของหลักการ divide & conquer คือ ปัญหาย่อยซึ่งซ้อนทับกันจะทำให้เสียเวลาในการหาคำตอบเนื่องจากจะต้องหาคำตอบเดิมๆ หลายครั้ง การโปรแกรมเชิงพลวัตร Dynamic Programming หรือ DP นำเสนอเทคนิคการออกแบบอัลกอริทึม โดยมีแนวคิดในการแบ่งปัญหาออกเป็นปัญหาย่อยและใช้วิธีการจำคำตอบ (memorization) เพื่อลดการคำนวณที่ซ้ำซ้อนประสิทธิภาพของเทคนิค DP จึงมาจากวิธีการจำคำตอบของปัญหาย่อยที่เตรียมไว้ล่วงหน้า สำหรับ ปัญหาขนาดใหญ่การโปรแกรมเชิงพลวัตร = substructure หรือ recursion + reuse (memorization) หลักการ dynamic programming เป็นวิธีการแก้ปัญหาที่มีประสิทธิภาพสูง เมื่อเทียบกับ divide & conquer และ brute force สำหรับ Dynamic programming จะใช้วิธีการแบ่งปัญหาออกเป็นปัญหาย่อย เพื่อหาคำตอบและอาศัยวิธีการจดจำคำตอบ (memorization) โดยเริ่มจากปัญหาย่อยที่เล็กที่สุด ประเด็นสำคัญ คือจำเป็นจะต้องออกแบบโครงสร้างของปัญหาย่อย (substructure) สำหรับการรวมคำตอบย่อย และวิธีการในการเตรียมคำตอบ ซึ่งมักขึ้นอยู่กับประเภทของปัญหา

ข้อแตกต่างของ Divide and Conquer คือ การแบ่งปัญหาออกเป็นปัญหาเล็กๆ และใช้ recursive เอาคำตอบมาต่อกันกลายเป็นคำตอบของปัญหาใหญ่ เช่น merge sort แบ่งปัญหาออกเป็นชิ้นๆ แล้วเอามาต่อกันจนครบ เอาคำตอบมาประกอบ

สำหรับ Dynamic Programming คือ การแก้ปัญหา โดยแบ่งปัญหาใหญ่ๆ ออกเป็นปัญหาเล็กๆ และแก้ปัญหาเหล่านี้ เอาผลลัพธ์ไปใส่ในตาราง เพื่อเอาคำตอบในตารางมาใช้ต่อไป การใส่ในตารางคือการทำ memorization เอาคำตอบไปเก็บแล้วเอามาคิดอีกครั้ง

8.2 ตัวอย่างการโปรแกรมเชิงพลวัตร

ตัวอย่างที่ 1 Fibonacci numbers

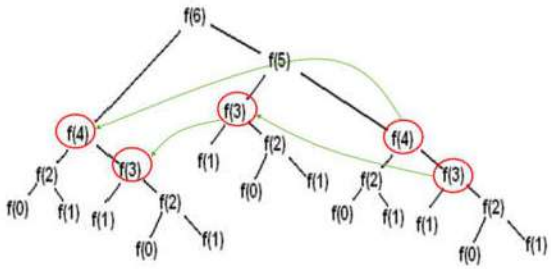
ลำดับเลขอนุกรมไฟโบนักชี (Fibonacci numbers) 1, 1, 2, 3, 5, 8, 13, 21, ... เราสามารถเขียนฟังก์ชันเพื่อคำนวณเลขไฟโบนักชีลำดับที่ n ได้

$$fib(n) = \begin{cases} 1, & n = 0,1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

Bruce Force

```
f(int n)
{
   if (n < 2) return 1;
   return f(n-2) + f(n-1);
}
```

n
2

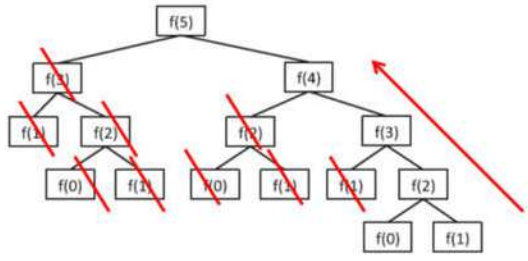


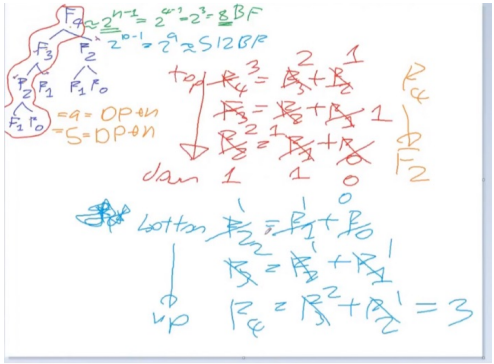
Dynamic Programming ใช้หน่วยความจำส่วนหนึ่งจดจำคำตอบที่เคยคำนวณมาก่อนหน้าเพื่อลดเวลาการทำงาน

```
int f(int n)
{
   int i;
   int F[100] = {0};
   F[0] = 0;
   F[1] = 1;
   for(i = 2; i<=n; i++)
      F[i] = F[i-1]+ F[i-2];
   return F[n];
}
f(20);
cout<< F[19] + F[18] <<endl;
//สำหรับ F20 = F19 + F18
```

Save เข้าใน Array

n





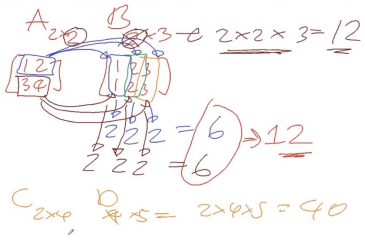
ปัญหาที่เหมาะกับ Dynamic Programming สามารถนิยามคำตอบของปัญหาในรูปของโครงสร้างย่อยซึ่งประกอบไปด้วยคำตอบของปัญหาย่อย (sub-structure) ตัวอย่างเช่น คำตอบของ f(n) สามารถนิยามเป็นโครงสร้างของปัญหาย่อยคือ f(n-1) + f(n-2) มีปัญหาย่อยทับซ้อนกัน (overlap sub-problems) เช่น เมื่อเราคำนวณ F(4) = F(3)+ F(2) และ F(3) = F(2) + F(1)

การเตรียมคำตอบของปัญหาย่อย (Memorization) เทคนิคของ dynamic programming ส่วนมาก มักจะใช้อาร์เรย์ (array) ในการจำคำตอบย่อยเพื่อลดเวลาในการคำนวณซ้ำ ซึ่งการออกแบบอาร์เรย์นั้นก็ต้องขึ้นอยู่กับปัญหา (อาจเป็น 1 มิติ หรือ 2 มิติ ก็ได้) ดังนั้น เวลาที่ใช้ในการหาคำตอบของปัญหาก็คือ การ update คำตอบในแต่ละช่องภายในอาร์เรย์ดังกล่าว กระบวนการสำรวจคำตอบมี 2 แบบ ดังนี้ 1) แบบ top-down 2) **แบบ bottom-up เริ่มจากปัญหาที่เล็กไปหาใหญ่ **นิยมใช้งานมากกว่า**

ตัวอย่างที่ 2 Matrix Chain Multiplication ✱ ออกสอบ

กำหนดให้เมตริกซ์ A มีขนาด n*m และเมตริกซ์ B มีขนาด m*p ดังนั้น C = AB จะได้ผลลัพธ์เป็นเมตริกซ์ C มีขนาด n*p ซึ่งหากใช้วิธีการคำนวณโดยตรง จำนวนครั้งในการคูณจะเท่ากับ n*m*p ครั้งหากขนาดของเมตริกซ์ AB เท่ากัน n*n จะต้องคูณกัน n^3 ครั้งสมมุติมีเมตริกซ์จำนวน n ชุด A1, A2, ..., An ขนาดมิติอาจไม่เท่ากัน จงหาลำดับการคูณเมตริกซ์ซึ่งทำให้ผลรวมของจำนวนการคูณน้อยที่สุด ตัวอย่างเช่น A1: [10*100] , A2: [100*5], A3:[5*50], A4:[50*1] ต้องการ A1 × A2 × A3 × A4 ซึ่งผลลัพธ์จะได้ [10*1]

Brute force การคิดจำนวนการคูณคือ 10 * 100 * 5 สำหรับ A1 * A2 และ 5 * 50 * 1 สำหรับ A3 * A4 การคูณ Matrix ลำดับมีผล ถ้าเอา A1 x A2 ไม่เท่ากับ A3 x A4 ก่อน คำตอบเหมือนกันแต่จำนวนรอบไม่เท่ากัน



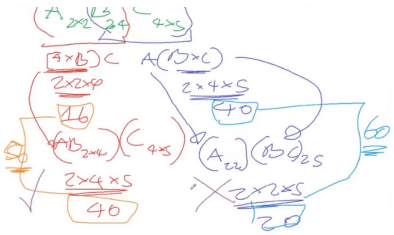
(A1(A2(**A3A4**)))
A34 = A3 × A4 = 250
A24 = A2 × A34 = 500
A14 = A1 × A24 = 1000
Total = 250 + 500 + 1000 = 1750

(((A1A2)A3)A4)
A12 = A1 × A2 = 5000
A13 = A12 × A3 = 2500
A14 = A13 × A4 = 500
Total = 5000 + 2500 + 500 = 8000

(A1((A2A3)A4))
A23 = A2 × A3 = 25000
A24 = A23 × A4 = 5000
A14 = A1 × A24 = 1000
Total = 25000 + 5000 + 1000 = 31000

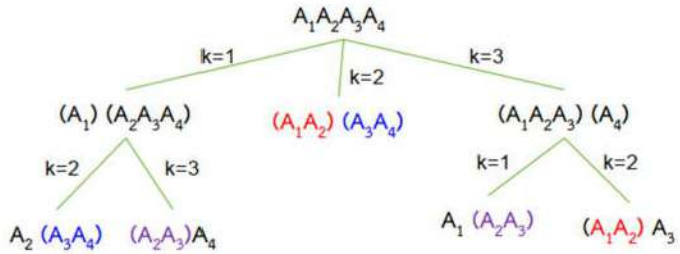
((A1A2)(A3A4))
A12 = A1 × A2 = 5000
A34 = A3 × A4 = 250
A14 = A12 × A34 = 50
Total = 5000 + 250 + 50 = 5300

((A1(A2A3))A4)
A23 = A2 × A3 = 25000
A13 = A1 × A23 = 50000
A14 = A13 × A4 = 500
Total = 25000 + 50000 + 500 = 75500



ถ้าเราเอา A3 * A4 ก่อน ตามด้วย A2 และ A1 จำนวนรอบ คือ 1750
ส่วน เอา A1*A2 ตามด้วย A3 และ A4 จำนวนรอบคือ 8000
ถ้า brute force คิดหมดแล้วเลือกเอา เอาที่น้อยที่สุด

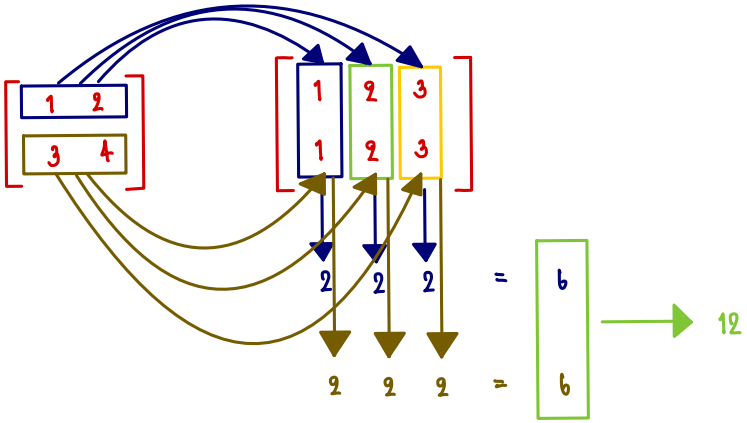
Dynamic Programming คำตอบของการคูณเมตริกซ์ $A_1.....A_n$ สามารถแบ่งเป็นโครงสร้างย่อย โดยการแบ่งลำดับการคูณของเมตริกซ์ออกเป็น 2 ส่วน คือ $(A_1...A_k)$ และ $(A_{k+1}...A_n)$ จากนั้นก็จะทำการรวมคำตอบที่ได้จากการคูณชุดเมตริกซ์ $A_1..A_k$ และ $A_{k+1}...A_n$ มารวมกันอีกครั้ง ซึ่งถ้าหากในแต่ละครั้งของการแบ่งชุดอาร์เรย์รับประกันว่าได้จำนวนการคูณน้อยที่สุด ก็จะทำให้คำตอบที่ต้องการน้อยที่สุดด้วย เรียกว่า optimal substructure จากรูปข้างล่าง ทั้งหมดมี 5 ในกรณี Brute force



A1: [10*100]
A2: [100*5]
A3: [5*50]
A4: [50*1]

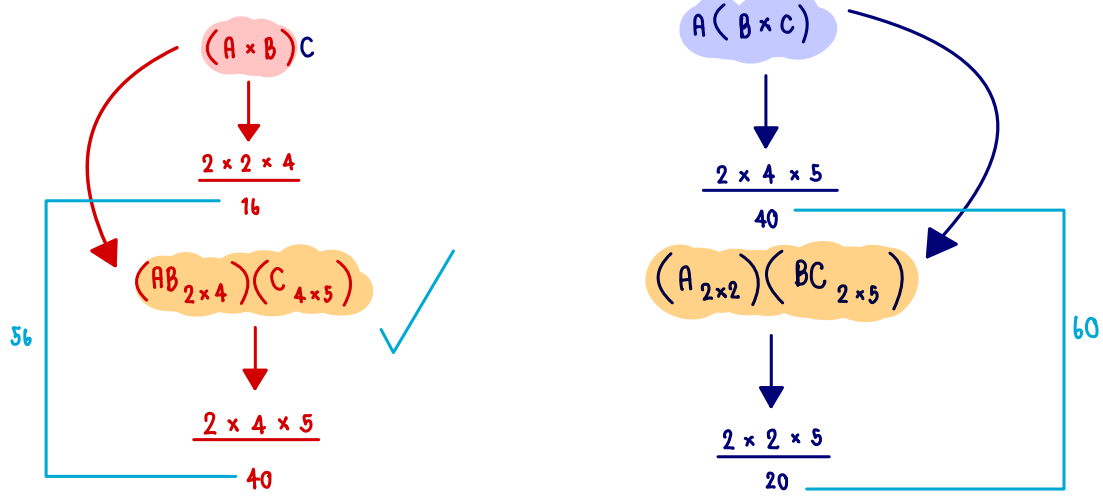
A1 × A2 × A3 × A4
[10*100] × [100*5] × [5*50] × [50*1]

$A_{2\times 2}$ $B_{2\times 3}$ → $2 \times 2 \times 3 = 12$

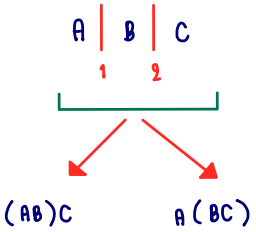


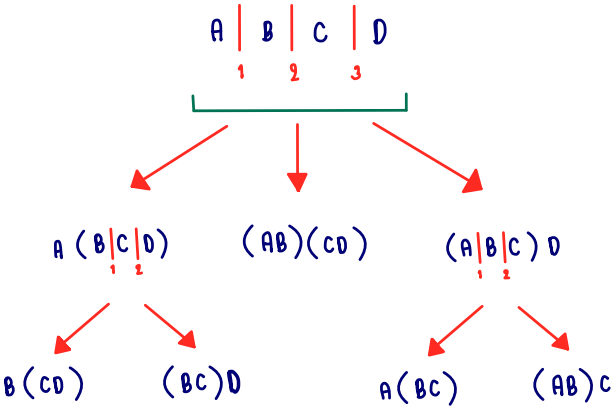
$C_{2\times 4}$ $D_{4\times 5}$ = $2 \times 4 \times 5$ = 40

$(A_{2\times 2}(B_{2\times 4})C_{4\times 5})$

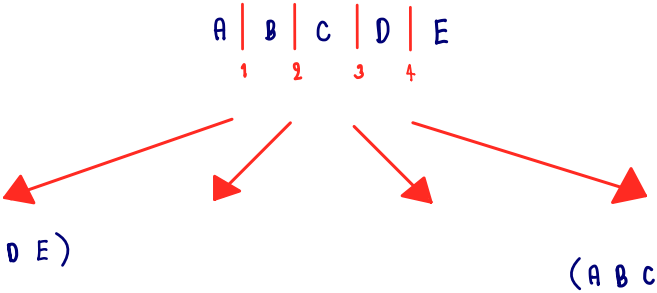


* เลขจำนวนรอบน้อย


A B C
1 2
(AB)C
A(BC)


A B C D
1 2 3
A(B C D)
1 2
(AB)(CD)
(A B C)D
1 2
B(CD)
(BC)D
A(BC)
(AB)C

* ข้าก


A B C D E
1 2 3 4
A(B C D E)
(A B C D)E

A_1 : [10 * 100] , A_2 : [100 * 5] , A_3 : [5 * 50] , A_4 : [50 * 1]

A234 A34

(A1 (A2 (A3 A4)))

A3 × A4 = A34

(5 , 50) (~~50~~ , 1) (5 , 1) = 250

A2 × A34 = A234

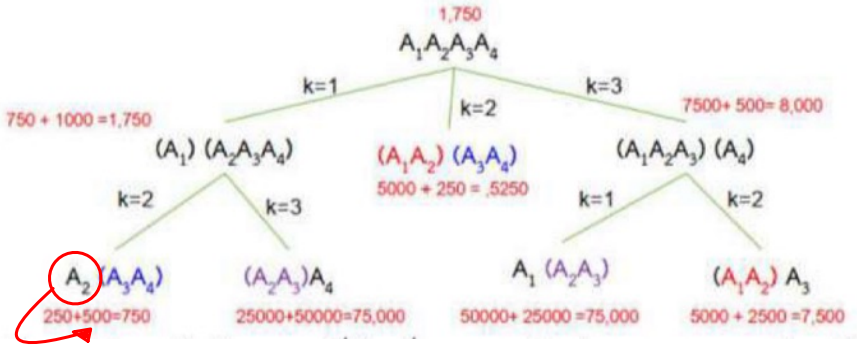
(100 , 5) (~~5~~ , 1) (100 , 1) = 500

1750

A1 × A2,3,4 = A1,2,3,4

(10 , 100) (~~100~~ , 1) (10 , 1) = 1000

คือผลรวม Brute Force

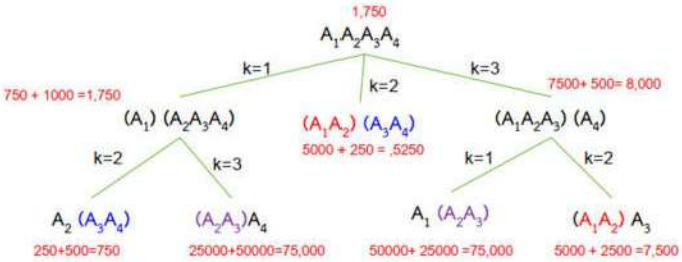


A_1 : [10 * 100] = 1000

A_2 : [100 * 5] = 500

A_3 : [5 * 50] = 250

A_4 : [50 * 1] = 50



จะเห็นได้ว่าหากเราเลือกคำตอบย่อยของชุดเมตริกซ์ที่ให้จำนวนการคูณที่น้อยที่สุดทุกครั้ง ก็จะทำให้ผลรวมของคำตอบย่อยได้น้อยที่สุดไปด้วย
คิดทุกกรณี จากข้างล่างเลือกตัวน้อยอย่างเดียวเท่านั้น เลือกเส้นที่น้อยที่สุด ขึ้นไป

กำหนดให้ $N_{i,j}$ เป็นจำนวนคูณที่น้อยที่สุดจากเมตริกซ์ $A_i....A_j$ สามารถเขียนเป็นสมการต่อเนื่องได้ดังนี้

$$N_{i,j} = \min_{i \le k < j} \{[N_{i,k}] + [N_{k+1,j}] + [d_i \times d_{k+1} \times d_{j+1}]\}$$

N_{14} = min{	n11 + n24 + 10*100*1,	เมื่อ k = 1
	n12 + n34 + 10*5*1,	เมื่อ k = 2
	n13 + n44 + 10*50*1 }	เมื่อ k = 3
N_{13} = min{	n11 + n23 + 10*100*50,	เมื่อ k = 1
	n12 + n33 + 10*5*50 }	เมื่อ k = 2
N_{24} = min{	n22 + n34 + 100*5*1 ,	เมื่อ k = 2
	n23 + n44 + 100*50*1 }	เมื่อ k = 3
N_{23} = n22 + n33 + 100*5*50		เมื่อ k = 2
N_{34} = n33 + n44 + 5*50*1		เมื่อ k = 3
N_{12} = n11 + n22 + 10*100*5		เมื่อ k = 3

	1	2	3	4
1	A1xA1=0	A1xA2	A1xA2xA3 มี 2 กรณี [A1xA2]xA3 A1x[A2xA3]	A1xA2xA3xA4 มี 3 กรณี * [A1xA2]x[A3xA4] * A1x[A2xA3xA4] * [A1xA2xA3]xA4
2		A2xA2=0	A2xA3	A2xA3xA4 มี 2 กรณี A2x[A3xA4] [A2xA3]xA4
3			A3xA3=0	A3xA4
4				A4xA4=0

```
int arr[] = { 10, 100, 5, 50, 1 };
int n = sizeof(arr) / sizeof(arr[0]);
int MatrixChainOrder(int i, int j)
{
    if (i == j){ return 0; }
    int min = 99999999999;
    int count;
    for (int k = i; k < j; k++)
    {
        count = MatrixChainOrder(i, k) + MatrixChainOrder(k + 1, j) + (arr[i - 1] * arr[k] * arr[j]);
        if (count < min) { min = count; } //MIN
    }
    return min; //RETURN MIN
}
int main()
{
    cout << MatrixChainOrder(1, n - 1);
}
```

int arr[] = { 10, 100, 5, 50, 1 }; มาจาก [10,100] [100,5] [5,50] [50,1] ทั้งหมด 4 อัน แต่ Array ขนาด 5 ช่อง เอาจุดที่คูณมาเชื่อมกัน สังเกตตารางจะเริ่มจาก 1 ไม่ใช่ 0

	1	2	3	4
1	1,1	ตำแหน่ง X = 5000	ตำแหน่ง Y	
2		2,2		
3			3,3	
4				4,4

ตำแหน่ง MatrixChainOrder(1, n - 1); เริ่มต้นที่ 1 (เริ่มต้นที่ 1 ไม่ใช่ 0 เพราะ สูตรพื้นฐาน n-1) กับ 4 (index สุดท้าย) ดูจากตาราง

ตำแหน่ง X คือตำแหน่งพื้นฐานที่สุด i=1,j=2
MatrixChainOrder(1, 1) = 0
MatrixChainOrder(2, 2) = 0
arr[i - 1] * arr[k] * arr[j] = arr[0] * arr[1] * arr[2] = 10 * 100 * 5 = 5000

ตำแหน่ง Y เกิดจากตำแหน่งของมันเอง + ตำแหน่ง i,k และ ตำแหน่ง k+1,j
จากสูตรพิกัด MatrixChainOrder(i, k) = 1,2
จากสูตรพิกัด MatrixChainOrder(k + 1, j) = 2,3

สำหรับ LOOP นี้ก่อนให้เกิดการท่องในตาราง for (int k = i; k < j; k++) { .... }
ก่อนเกิดการ ท่องใน ตาราง ดังนี้
i=1,k=1,j=4 → 1,1 = 0 / 1,4 ยังคำนวณไม่ได้ → FINAL
i=2,k=2,j=4 → 2,2 = 0 / 2,4 ยังคำนวณไม่ได้ → 2,4
i=3,k=3,j=4 → 3,3 = 0 4,4 = 0 → 0 + 0 + (3,4)
i=2,k=3,j=4
i=2,k=2,j=3 →2,2 = 0 3,3 = 0 → 0 + 0 + (2,3)
i=1,k=2,j=4
i=1,k=1,j=2 →1,1 = 0 2,2 = 0 → 0 + 0 + (1,2)
i=3,k=3,j=4 ซ้ำ
i=1,k=3,j=4
i=1,k=1,j=3 → 1,1 = 0 / 1,3 ยังคำนวณไม่ได้ → 1,3
i=2,k=2,j=3 ซ้ำ
i=1,k=2,j=3
i=1,k=1,j=2 ซ้ำ

การเตรียมคำตอบของปัญหา matrix chain

	1	2	3	4
1	0	N12	N13	N14
2		0	N23	N24
3			0	N34
4				0

A1: [10*100]

A2: [100*5]

A3: [5*50]

A4: [50*1]

	1	2	3	4
1	0	5000		
2		0		
3			0	
4				0

$N_{12} = (n_{11} = 0) + (n_{22} = 0) + (10*100*5 = A1*A2$ เอามาคูณกันปกติ) = 5000 (ขนาด 10 * 5)

	1	2	3	4
1	0	5000		
2		0	25000	
3			0	
4				0

$N_{23} = n_{22} + n_{33} + 100*5*50 = 25000$ (ขนาด 100 * 50)

	1	2	3	4
1	0	5000		
2		0	25000	
3			0	250
4				0

$N_{34} = n_{33} + n_{44} + 5*50*1 = 250$ (ขนาด 5 * 1)

	1	2	3	4
1	0	5000	7500	
2		0	25000	
3			0	250
4				0

N_{13} = min

{

~~0 + 25000 + 10*100*50 = 75000~~ ~~เมื่อ k = 1 → เอา A1 * (A2A3) (ขนาด 10 * 50)~~

5000 + 0 + 10*5*50 = 7500 เมื่อ k = 2 → เอา (A1A2) * A3) (ขนาด 10 * 50)

}

	1	2	3	4
1	0	5000	7500	
2		0	25000	750
3			0	250
4				0

N_{24} = min

{

0 + 250 + 100*5*1 เมื่อ k = 2

~~25000 + 0 + 100*50*1 เมื่อ k = 3~~

}

	1	2	3	4
1	0	5000	7500	1750
2		0	25000	750
3			0	250
4				0

N_{14} = min

{

0 + 750 [A2A3A4] + 10*100*1 เอา A1 * (A2A3A4) เมื่อ k = 1

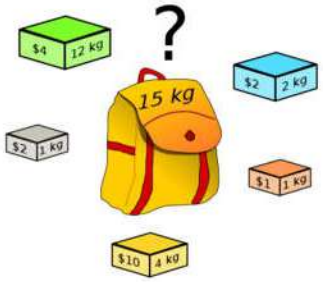
~~5000 [A1A2] + 250 [A3A4]+ 10*5*1 เมื่อ k = 2~~

~~7500 [A1A2A3] + 0 + 10*50*1 เอา (A1A2A3) * (A4) เมื่อ k = 3~~

} = 1,750

* ตกสอบ

ตัวอย่างที่ 3 0/1 knapsack problem ถุงใส่ของรับน้ำหนัก Wt กิโลกรัม มีของ n ชิ้น แต่ละชิ้นมีน้ำหนัก (wi) และมูลค่า (vi) ที่แตกต่างกัน หาวิธีเลือกของใส่ถุงเพื่อให้มีมูลค่ารวมมากที่สุด และน้ำหนักรวมไม่เกิน Wt โดยถุงมีความจุจำกัด ใส่ของให้ได้เงินมากที่สุด และถุงไม่ขาด



Bag = 7

item	wi	vi
1	1	1
2	3	4
3	4	5
4	5	7

$$\max \sum_{i \in T} v_i \quad subject\ to \quad \sum_{i \in T} w_i \le W_t$$

เลือก item 1,2 : value = 5
เลือก item 1,3 : value = 6
เลือก item 4 : value = 7
เลือก item 2,3 : value = 9 ได้ wi คือ 7 Optimal solution
เลือก item 2,4 : value = 11 ไม่ได้ wi เกิน 8

Brute force จากนิยามของปัญหา พบว่าทุกไอเท็มสามารถพิจารณาเลือกได้ 2 ทาง คือ

1. ถ้าน้ำหนักของไอเท็มไม่เกินน้ำหนักของถุงที่ยังรับได้ เราก็จะใส่ไอเท็มเข้าไปในถุง ดังนั้นมูลค่ารวมก็จะเพิ่มขึ้น
2. ถ้าน้ำหนักเกินก็ไม่ใส่ไอเท็มเข้าไปในถุง มูลค่ารวมก็จะเท่ากับไอเท็มที่ถูกใส่ก่อนหน้า ดังนั้นหากรับประกันได้ว่ามูลค่ารวมของไอเท็มที่ใส่ก่อนหน้ามากที่สุด การใส่ไอเท็มนี้เข้าไปก็จะทำให้ได้ผลรวมของมูลค่ามากที่สุดด้วย

Item	1	2	3	4	
1	0	0	0	0	ไม่เลือก
2	1	0	0	0	เลือก 1 ได้
3	0	1	0	0	เลือก 2 ได้
...	..	..	..	..	..
16	1	1	1	1	ไม่ได้

Brute force ทั้งหมด 2^4 = 16 ไล่คำตอบที่เป็นไปได้ทั้งหมด
ถ้ามี 100 ชิ้น คือ 1267650600228229401496703205376

Dynamic Programming (คิดแบบคนคิด)

	เรียง Weight จากน้อยไปหามาก		Max weight						
Item	Weight	Value	1	2	3	4	5	6	7
1	1	1	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1
2	3	4	1 เลือกอัน 1 Value = 1	1 เลือกอัน 1 Value = 1	4 เลือกอัน 2 Value = 4	5 เลือกอัน 1 กับ 2 Value = 5	5 เลือกอันที่ 1 กับ 2 Value = 5	5 เลือกอันที่ 1 กับ 2 Value = 5	5 เลือกอันที่ 1 กับ 2 Value = 5
3	4	5	1 เลือกอัน 1	1 เลือกอัน 1	4 เลือกอัน 2	5 เลือกอัน 3	6 เลือกอัน 3+1 ** เลือกให้ได้มากที่สุด	6 เลือกอัน 3+1	9 เลือกอัน 3+2
4	5	7	1 เลือกอัน 1	1 เลือกอัน 1	4 เลือกอัน 2	5 เลือกอัน 3	7 เลือกอัน 5	8 เลือกอัน 4+1	9 เลือกอัน 3+2 กรณี Brute force ถ้า choice มันเยอะต้องมานั่งใส่ set 0 กับ 1 ทั้งหมดจะมีค่ามากกว่าการแบ่งเป็นส่วนๆ เช่น ตาราง 100 x 100

Dynamic Programming (คิดแบบสมการ)

Max_Weight = 7

Value = i : 1 4 5 7

Weight = j : 1 3 4 5

$$M_{ij} = \begin{cases} \max\{V_i + M_{i-1,j-w_i}, M_{i-1,j}\}, & w_i < j \\ M_{i-1,j}, & otherwise \end{cases}$$

กำหนดให้ M_{ij} หมายถึง มูลค่ารวมที่มากที่สุดเมื่อใส่ของชิ้นที่ i ลงไปในถุงที่รับน้ำหนักได้ j ดังนั้น

เริ่มต้นจากการเติมค่า 0 ให้กับคอลัมน์ 0 และแถว 0

	0	1	2	3	4	5	6	7
0	M_{00}	M_{01}	M_{02}	M_{03}	M_{04}	M_{05}	M_{06}	M_{07}
1	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}	M_{15}	M_{16}	M_{17}
2	M_{20}	M_{21}	M_{22}	M_{23}	M_{24}	M_{25}	M_{26}	M_{27}
3	M_{30}	M_{31}	M_{32}	M_{33}	M_{34}	M_{35}	M_{36}	M_{37}
4	M_{40}	M_{41}	M_{42}	M_{43}	M_{44}	M_{45}	M_{46}	M_{47}

i \ j	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
(1, 1)	0							
(4, 3)	0							
(5, 4)	0							
(5, 7)	0							

Wt = 7

V : 1 4 5 7

W: 1 3 4 5

M_{11} = max { 1 + [$M_{0,0}$ = 0] , [$M_{0,1}$ = 0] } = 1

M_{12} = max { 1 + $M_{0,1}$, $M_{0,2}$} = 1

M_{13} = max { 1 + $M_{0,2}$, $M_{0,3}$} = 1

....

i \ j	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
(1, 1)	0	1	1	1	1	1	1	1
(4, 3)	0							
(5, 4)	0							
(5, 7)	0							

Wt = 7

V : 1 4 5 7

W: 1 3 4 5

M_{21} = M_{11} = 1

M_{22} = M_{12} = 1

M_{23} = {4 (3KG ใส่ 4 ได้) + [M_{10} = 0] , M_{13} = 1} = 4

M_{24} = {4 (3KG ใส่ 4 ได้) + [M_{11} = 1] , M_{14} = 1} = 5

....

i \ j	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
(1, 1)	0	1	1	1	1	1	1	1
(4, 3)	0	1	1	4	5	5	5	5
(5, 4)	0							
(7, 5)	0							

Wt = 7

V : 1 4 5 7

W: 1 3 4 5

....

M_{47} = max

{

7 + M_{32} (ใส่จากช่องข้างหน้าที่ใส่ได้ 0 ถึง 7 ช่องก่อนหน้า

ถ้าใส่ไม่ได้ให้หยุด เช่น

5 kg + 0 kg ได้ 7 + $M_{3,0}$ value = 1

5 kg + 1 kg ได้ 7 + $M_{3,1}$ value = 1

5 kg + 1 kg ได้ 7 + $M_{3,2}$ value = 1

5 kg + 4 kg ถุงขาดหยุด

จึงได้มาเป็น 7 + $M_{3,2}$ value = 1 ได้ค่าเป็น 8

เปรียบเทียบกับกรณีเดิม คือ $M_{3,7}$ มี value คือ 9

โดย 3 กับ 7 เลือกชิ้นที่ 3 กับ ชิ้นที่ 2

) ,

M_{37}

} = max { 7+1, 9 } = 9

i \ j	0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0
(1, 1)	0	1	1	1	1	1	1	1
(4, 3)	0	1	1	4	5	5	5	5
(5, 4)	0	1	1	4	5	6	6	9
(7, 5)	0	1	1	4	5	7	8	9

Val	Wt	Item	Max Weight 0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1
4	3	2	0	1	1	4	5	5	5	5
5	4	3	0	1	1	4	5	6	6	9
7	5	4	0	1	1	4	5	7	8	9

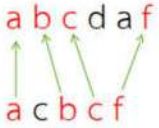
`int max(int a, int b) { if(a > b){ return a; }else{ return b; } }`	หาค่า max
`int weight[] = {1, 3, 4, 5};`	อาร์เรย์ weight
`int value[] = {1, 4, 5, 7};`	อาร์เรย์ value
`int row = 4;`	จำนวนวัตถุ
`int col = 7;`	Max weight
`int array[5][8];`	row+1, col+1 เพิ่มแถว 0
ใน main()	
`for(int i=0 ; i <= row ; i++)`	Row
`{`	
`for(int j=0 ; j <= col ; j++)`	Weight
`{`	
`if (i == 0 || j == 0) {array[i][j] = 0; }`	กรณีแถว 0
`else if ( weight[i - 1] <= j )`	กรณีนี้ สามารถเติมข้อมูล j คือ weight สำหรับช่องบน เช่น แถว ที่ 1 weight = 1 ถ้า column เป็น 2 สามารถใส่ได้ เนื่องจากเริ่มที่ 0 ดังนั้นช่องนั้นต้องลบ 1 จริงคือ ค่าของช่องแถวนั้น เช่นแถวที่ 2 คือช่องที่ value = 4 Array[i-1] ค่าของแถวข้างบน ที่ตำแหน่ง"หลักของมัน" - น้ำหนักของช่องนั้นๆ เช่น แถวที่ 2 น้ำหนักเท่ากับ หลักที่ 3 ต้องการ 3 KG วัตถุลบแล้วเหลือ 0 เอาช่องที่ 0 มาบวก เปรียบเทียบกับช่องข้างบน เช่น แถวที่ 2 เปรียบกับ แถวที่ 1 ใน column เดียวกัน
`{`	

`array[i][j] = max( value[i - 1] +`
`array[i - 1][ j - weight[i - 1] ],`
`array[i - 1][j] );`	

`}`	
`else { array[i][j] = array[i - 1][j]; }`	กรณีนี้ สามารถเติมข้อมูล แถว ที่ 2 weight = 3 ถ้า column เป็น 2 สามารถใส่ได้ แต่ถ้า column เป็น 1 เกินแล้วใส่ไม่ได้เอาจากข้างบนลงมา
`}`	
`}`	

ตัวอย่างที่ 4 Longest Common Subsequence (LCS)

กำหนดให้ A และ B เป็นสายอักขระ (string) ซึ่งอาจมีความยาวไม่เท่ากัน ต้องการหา**ลำดับร่วมที่ยาวที่สุด**ของ A และ B ตัวอย่างเช่น จากตัวอย่างด้านล่าง ลำดับร่วมที่ยาวที่สุดคือ 4



brute force คือการเช็คทุกลำดับร่วมที่เป็นไปได้ ซึ่งจะเห็นได้ว่ามีการทำงานที่ซ้ำซ้อนและใช้เวลาในการประมวลผลค่อนข้างนาน

String A = "acbaed";
String B = "abcadf";

String A	a	c	b	a	e	d
String B	a	b	c	a	d	f

Longest Common Subsequence(LCS): acad, Length: 4

คือเอา String A ตัว a ไปไล่ check String B ได้แน่นนอน เพราะใน String B มี a อยู่ สำหรับการ check

a	b	c	a	d	f	ผ่าน
✓						

คือเอา String A ตัว ac ไปไล่ check String B ได้ สำหรับการ check

a	b	c	a	d	f	ผ่าน
✓		✓				

คือเอา String A ตัว acb ไปไล่ check String B ได้หรือไม่ สำหรับการ check

a	b	c	a	d	f	ไม่ผ่าน หลัง c ไม่มี b เลย
✓		✓				

คือเอา String A ตัว aca ไปไล่ check String B ได้หรือไม่ สำหรับการ check

a	b	c	a	d	f	ผ่าน
✓		✓	✓			

คือเอา String A ตัว acae ไปไล่ check String B ได้หรือไม่ สำหรับการ check

a	b	c	a	d	f	ไม่ผ่าน หลัง a ไม่มี e เลย
✓		✓	✓			

คือเอา String A ตัว acad (หมดแล้ว) ไปไล่ check String B ได้หรือไม่ สำหรับการ check

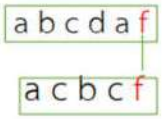
a	b	c	a	d	f	ผ่าน
✓		✓	✓	✓		

Dynamic Programming

Optimal substructure of LCS คือ กำหนดให้สายอักขระ X และ Y มีความยาวคือ m และ n ตามลำดับ ให้ L(m, n) แทนความยาวของลำดับร่วมที่มากที่สุดระหว่าง X และ Y ซึ่งสามารถพิจารณาจาก 2 กรณี ดังนี้

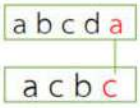
<u>กรณีที่ 1</u> <u>หาก Xm = Yn</u> แล้ว

L(m, n) = 1 + L(m-1, n-1)



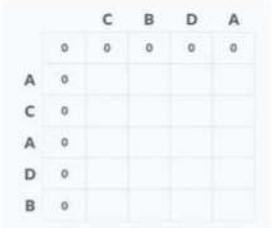
<u>กรณีที่ 2</u> <u>หาก X[m] < หรือ > Y[n]</u> แล้ว

L(m, n) = max{ L(m, n-1) , L(m-1, n) }

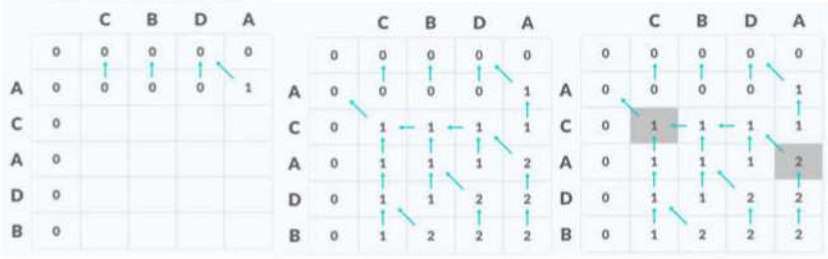


Dynamic Programming (คิดแบบคน)

1) สร้างตารางตามความยาวของ String ทั้ง 2 ตัว ใส่ 0 row และ column แรก



2) ถ้าตัวอักษรไหนแถวกับหลักซ้ำกันให้ +1 จากแนวเฉียงเติมให้เต็มทั้งตาราง เอาตัวแรกของการเปลี่ยนค่าเป็นคำตอบ สำหรับแนวราบให้ซ้ำค่านั้นทั้งแถวจนกว่าจะเจอตัวที่เปลี่ยนค่าจากแนวเฉียง



Dynamic Programming (คิดแบบสมการ)

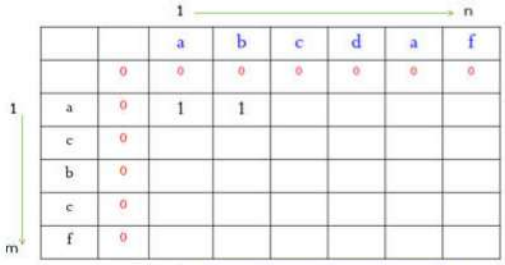
$$L(m,n) = \begin{cases} 1 + L(m-1, n-1), & X[m] = Y[n] \\ \max\{L(m, n-1), L(m-1, n)\}, & otherwise \end{cases}$$

a b c d a f

a c b c f

L(1,1) = 1 + L(0,0) = 1

L(1,2) = max { L(1,1) , L(0,1) } = 1

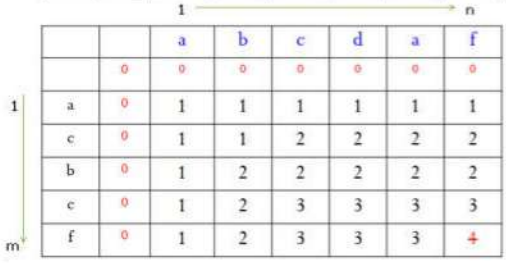


a b c d a f

a c b c f

L(2,1) = max { L(2, 0) , L(1,1) } = 1

L(2,3) = 1 + L(1, 2) = 1



char X[] = "acbcf";	ข้อความแรก
char Y[] = "abcdaf";	ข้อความที่สอง
int s_x = 5;	ขนาดของข้อความแรก
int s_y = 6;	ขนาดของข้อความที่สอง
int array[10][10];	Array ขนาดใหญ่จองๆ ไว้ก่อน
int max(int a, int b){if(a > b){ return a; }else{ return b; }}	หาค่ามากสุด
ในฟังก์ชั่น main	
for(int i=0 ; i <= s_x ; i++)	ขนาดข้อความแรก โดยเริ่มตั้งแต่ 0
{	
for(int j=0 ; j <= s_y ; j++)	ขนาดข้อความที่สอง โดยเริ่มตั้งแต่ 0
{	
if (i == 0 || j == 0){ array[i][j] = 0; }	แถวและคอลัมแรกเป็น 0
else if (X[i-1] == Y[j-1])	ถ้าข้อความเป็นตัวอักษรเดี่ยวกัน ต้อง -1 เพราะว่า ข้อความ
{	Index ของ char X หรือ char Y แรกเริ่มที่ 0 แต่ใน loop จะเริ่มที่ 1
array[i][j] = 1 + array[i-1][j-1];	ให้เพิ่ม 1 และเอาค่าเฉลี่ยๆ มาบกวเพิ่ม
}	
else	
{	
array[i][j] = max(array[i][j-1] , array[i-1][j]);	ถ้าเท่าเดิมให้เอาด้านบน หรือด้าน ที่มากที่สุดใส่ในช่องนั้น
}	
}	
}	

ตัวอย่างที่ 5 Game Strategy (coin-in-a-line)

กำหนดให้มี n เหรียญวางเรียงลำดับ โดยที่ n เป็นเลขคู่และมีมูลค่าแตกต่างกัน ผู้เล่น 2 คน คือ Alice และ Bob จะต้องผลัดกันหยิบเหรียญให้ได้มูลค่ารวมมากที่สุด โดยผู้เล่นแต่ละคนสามารถเลือกหยิบเหรียญได้เฉพาะเหรียญที่อยู่ปลายแถว (ซ้าย-ขวา) เท่านั้น มีวิธีหยิบเหรียญเหล่านี้ยังไงให้ได้มูลค่ารวมมากที่สุด

ตัวอย่างที่ 5.1

coins [] = { 6, 9, 1, 2, 16, 8 }

trial 1: (players will pick the best option available for them)

coins [] = { 6, 9, 1, 2, 16, 8} , Alice picks 8

coins [] = { 6, 9, 1, 2, 16}, Bob picks 16

coins [] = { 6, 9, 1, 2}, Alice picks 6

coins [] = { 9, 1, 2}, Bob picks 9

coins [] = {1, 2}, Alice picks 2

coins [] = {1}, Bob picks 1

Alice: 8+6+2 = 16

Bob: 16+9+1 = 26 => Alice Lost

ตัวอย่างที่ 5.2

coins [] = { 6, 9, 1, 2, 16, 8 }

trial 2: (Alice thinks about Bob's move, Will discuss the strategy in solution)

coins [] = { 6, 9, 1, 2, 16, 8} , Alice picks 6

coins [] = { 9, 1, 2, 16, 8}, Bob picks 9

coins [] = { 1, 2, 16, 8}, Alice picks 1

coins [] = 2, 16, 8}, Bob picks 8

coins [] = {2, 16}, Alice picks 16

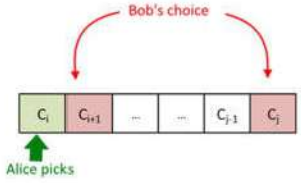
coins [] = {2}, Bob picks 2

Alice: 6+1+16 = 23

Bob: 9+8+2 = 19 => Alice Won

สมมติว่ามีผู้เล่น A และ B และ เหรียญ $C_i\ C_{i+1}\ C_{i+2}$ ..... $C_{j-2}\ C_{j-1}\ C_j$ กำหนดให้ M_{ij} คือมูลค่ารวมที่มากที่สุดของเหรียญที่ถูกหยิบ

Alice Picks the i^{th} coin (from starting)

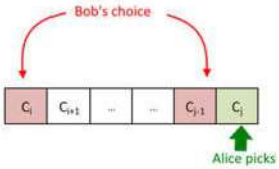


โดย A หาก A เริ่มเล่นก่อน A จะเลือก C_i กรณีหยิบตัวแรก

B สามารถเลือก Ci+1 ซึ่งจะทำให้ A จะมูลค่ารวมคือ Vi + Mi+2,j หรือ

B สามารถเลือก Cj ซึ่งจะทำให้ A จะมูลค่ารวมคือ Vi + Mi+1,j-1

Alice Picks the j^{th} coin (from the end)



A จะเลือก C_j กรณีหยิบตัวสุดท้าย

B สามารถเลือก Ci ซึ่งจะทำให้ A จะมูลค่ารวมคือ Vj + Mi+1,j-1 หรือ

B สามารถเลือก Cj-1 ซึ่งจะทำให้ A จะมูลค่ารวมคือ Vj + Mi,j-2

F(i, j) = Max

(Vi (เงินที่ได้จากการหยิบอันแรก) + min(F(i+2, j) [อีกคนหยิบ i ไป] , F(i+1, j-1) [อีกคนหยิบ j ไป] ทำให้อีกคนน้อยสุด),

Vj (เงินที่ได้จากการหยิบอันสุดท้าย) + min(F(i+1, j-1) [อีกคนหยิบ i ไป] , F(i, j-2) [อีกคนหยิบ j ไป] ทำให้อีกคนน้อยสุด)) เราต้องได้มากสุด

$$M_{ij} = max \begin{cases} \min\{M_{i+1,j-1} + V_j\ ,\ M_{i,j-2} + V_j\}, & player => C_j \\ \min\{M_{i+2,j} + V_i\ ,\ M_{i+1,j-1} + V_i\}, & player => C_j \end{cases}$$

สมมติว่ามีเหรียญทั้งหมด 4 เหรียญ ได้แก่ 3 9 1 2 นักศึกษาเป็นฝ่ายเล่นก่อน

กำหนดค่าเริ่มต้นจากเหรียญ 1 เหรียญ ดังนั้นหาก A หยิบก่อน A ก็จะได้มูลค่าเหรียญนั้นไป และ B จะได้ 0 เนื่องจากไม่มีเหรียญจะหยิบ

	1	2	3	4
1	(3,0)			
2		(9,0)		
3			(1,0)	
4				(2,0)

เมื่อมีเหรียญเพิ่มเป็น 2 เหรียญ A ก็จะเลือกหยิบ 1 เหรียญ หลังจากนั้น B ก็หยิบเหรียญที่เหลือ

	1	2	3	4
1	(3,0)	(9, 3)		
2		(9, 0)	(9, 1)	
3			(1, 0)	(2, 1)
4				(2,0)

	1	2	3	4
1	(3,0)	(9, 3)	(4, 9)	(11, 4)
2		(9, 0)	(9, 1)	(10, 2)
3			(1, 0)	(2, 1)
4				(2,0)

	1 | value = 3	2 | value = 9	3 | value = 1	4 | value = 2
1	3,0 คนแรก P1 = 3 เลือก 3 คนที่สอง P2 = 0 ไม่ได้เลือก **** รอบแรก P1 หยิบก่อน**	ตัวแรกคิดจาก ตรงนี้ 3 9 จากตรงนี้เลือกได้ 2 เหรียญคือ 3 กับ 9 9 = 9 + { 0 → ตัวที่ 2 ของ [1,1] } 3 = 3 + { 0 → ตัวที่ 2 ของ [2,2] } เลือกเอามากสุด จึงเลือก 9 ตัวสองคิดจาก โดยเลือกตัวที่น้อยที่สุดจากช่องแรก ในกรณีคือ 3 กับ 9 เลือก 3 **ผลลัพธ์ คือ 9, 3** **** รอบสอง P2 หยิบ 9 และ P1 หยิบ 3**	ตัวแรกคิดจาก ตรงนี้ 3 9 1 จากตรงนี้เลือกได้ 2 เหรียญคือ 3 กับ 1 ถ้าเลือก 3 ก่อนหน้าไม่ได้เลือก 3 4 = 3 + { 1 → ตัวที่ 2 ของ [2,3] } มาจาก 9, 1 คือ P1 เลือก 1 และ P2 เลือก 9 ถ้าเลือก 1 ก่อนหน้าไม่ได้เลือก 1 4 = 1 + { 3 → ตัวที่ 2 ของ [1,2] } มาจาก 9, 3 คือ P1 เลือก 3 และ P2 เลือก 9 โดยเลือกตัวที่มากที่สุด ในกรณีคือ 4 กับ 4 เลือก 4 ตัวสองคิดจาก ตัวแรกของช่อง [1,2] และ [2,3] โดยเลือกตัวน้อยสุด 1,2 = 9 และ 2,3 = 9 จึงเลือก 9 **ผลลัพธ์ คือ 4, 9** **** รอบสาม P1 หยิบ**	ตัวแรกคิดจาก ตรงนี้ 3 9 1 2 จากตรงนี้เลือกได้ 2 เหรียญคือ ถ้าเลือก 3 ก่อนหน้าไม่ได้เลือก 3 5 = 3 + { 2 → ตัวที่ 2 ของ [2,4] } ถ้าเลือก 2 ก่อนหน้าไม่ได้เลือก 2 11 = 2 + { 9 → ตัวที่ 2 ของ [1,3] } เลือกเอามากสุด จึงเลือก 11 ตัวสองคิดจาก สูตร ตัวแรกของช่อง [2,4] และ [1,3] โดยเลือกตัวน้อยสุด 2,4 = 10 และ 1,3 = 4 จึงเลือก 4 **ผลลัพธ์ คือ 11, 4** **** รอบสาม P2 หยิบ**
2		9,0 คนแรก P1 = 9 เลือก 9 คนที่สอง P2 = 0 ไม่ได้เลือก	ตัวแรกคิดจาก ตรงนี้ 9 1 จากตรงนี้เลือกได้ 2 เหรียญคือ 9 กับ 1 9 = 9 + { 0 → ตัวที่ 2 ของ [3,3] } 1 = 1 + { 0 → ตัวที่ 2 ของ [2,2] } เลือกเอามากสุด จึงเลือก 9 ตัวสองคิดจาก สูตร โดยเลือกตัวที่น้อยที่สุดจากช่องแรก ในกรณีคือ 9 กับ 1 เลือก 1 **ผลลัพธ์ คือ 9, 1**	ตัวแรกคิดจาก ตรงนี้ 9 1 2 จากตรงนี้เลือกได้ 2 เหรียญคือ 9 กับ 2 ถ้าเลือก 2 ก่อนหน้าไม่ได้เลือก 2 3 = 2 + { 1 → ตัวที่ 2 ของ [2,3] } มาจาก 9, 1 คือ P1 เลือก 1 และ P2 เลือก 9 ถ้าเลือก 9 ก่อนหน้าไม่ได้เลือก 9 4 = 9 + { 1 → ตัวที่ 2 ของ [3,4] } มาจาก 2,1 คือ P1 เลือก 3 และ P2 เลือก 9 โดยเลือกตัวที่มากที่สุด ในกรณีคือ 10 กับ 3 เลือก 10 ตัวสองคิดจาก ตัวแรกของช่อง [2,3] และ [3,4] โดยเลือกตัวน้อยสุด 2,3 = 9 3,4 = 2 จึงเลือก 2 **ผลลัพธ์ คือ 10, 2** **** รอบสาม P1 หยิบ**
3			1,0 คนแรก P1 = 1 เลือก 1 คนที่สอง P2 = 0 ไม่ได้เลือก	ตัวแรกคิดจาก ตรงนี้ 1 2 จากตรงนี้เลือกได้ 2 เหรียญคือ 1 กับ 2 2 = 2 + { 0 → ตัวที่ 2 ของ [3,3] } 1 = 1 + { 0 → ตัวที่ 2 ของ [4,4] } เลือกเอามากสุด จึงเลือก 2 ตัวสองคิดจาก สูตร โดยเลือกตัวที่น้อยที่สุดจากช่องแรก ในกรณีคือ 2 กับ 1 เลือก 1 **ผลลัพธ์ คือ 2, 1**
4				2,0 คนแรก P1 = 2 เลือก 2 คนที่สอง P2 = 0 ไม่ได้เลือก

สมมติว่ามีเหรียญทั้งหมด 4 เหรียญ ได้แก่ 3 9 1 2

สรุป P1 หยิบ 1 | P2 หยิบ 9 | P1 หยิบ 3 | P2 หยิบ 2

รอบที่ 1	รอบที่ 2	รอบที่ 3	รอบที่ 4
0 0	0 1	0 2	0 3
1 1	1 2	1 3	
2 2	2 3		
3 3			

	int arr[] = { 3, 9, 1, 2 };			
	1 | value = 3 Index=0	2 | value = 9 Index=1	3 | value = 1 Index=2	4 | value = 2 Index=3

1
Index=0
	3	9 → 0 1
		จากตรงนี้เลือกได้ 2 เหรียญคือ 3 กับ 9
		9 = 9 + { 0 → ตัวที่ 2 ของ [1,1] }
		3 = 3 + { 0 → ตัวที่ 2 ของ [2,2] }
		เลือกเอามากสุด จึงเลือก 9
		table[i][j] =
		max(
		arr[0] = 3 + min(x, y),
		arr[1] = 9 + min(y, z)
		);
		
		int x = 0;
		if((i + 2) → 2 <= j → 1)
		{
		x = table[i + 2][j]; ตำแหน่ง 2,1
		}
		ผิดเงื่อนไขได้คำตอบเป็น 0
		
		int y = 0;
		if((i + 1) → 1 <= (j - 1) → 0)
		{
		y = table[i + 1][j-1]; ตำแหน่ง 1,0
		}
		ผิดเงื่อนไขได้คำตอบเป็น 0
		
		int z = 0;
		if(i → 0 <= (j - 2) → -1)
		{
		z = table[i][j - 2]; ตำแหน่ง 0, -1
		}
		ผิดเงื่อนไขได้คำตอบเป็น 0
		
		max(
		arr[0] = 3 + 0
		arr[1] = 9 + 0
		);
			4 → i = 0 j = 2
			จากตรงนี้เลือกได้ 2 เหรียญคือ 3 กับ 1
			ถ้าเลือก 3 ก่อนหน้าไม่ได้เลือก 3
			4 = 3 + { 1 → ตัวที่ 2 ของ [2,3] }
			มาจาก 9, 1 คือ P1 เลือก 1 และ P2 เลือก 9
			ถ้าเลือก 1 ก่อนหน้าไม่ได้เลือก 1
			4 = 1 + { 3 → ตัวที่ 2 ของ [1,2] }
			มาจาก 9, 3 คือ P1 เลือก 3 และ P2 เลือก 9
			โดยเลือกตัวที่มากที่สุด
			ในกรณีคือ 4 กับ 4 เลือก 4
			
			table[i][j] =
			max(
			arr[0] = 3 + min(x, y),
			arr[1] = 1 + min(y, z)
			);
			
			int x = 0;
			if((i + 2) →2 <= j →2)
			{
			x = table[2][2]; ตำแหน่ง 2,0 → 1
			}
			
			int y = 0;
			if((i + 1) → 1 <= (j - 1) → 1)
			{
			y = table[1][1]; ตำแหน่ง 1,1 → 9
			}
			
			int z = 0;
			if(i → 0 <= (j - 2) → 0)
			{
			z = table[0][0]; ตำแหน่ง 0, 0 → 3
			}
			arr[0] = 3 + min(1, 9), = 4
			arr[1] = 1 + min(9, 3) = 4
				11

2 Index=1	0	9	9	10
3 Index=2	0	0	1	2
4 Index=3	0	0	0	2

แบบฝึกหัด

1) จงเขียนโปรแกรมแก้ปัญหา Matrix Chain Multiplication ด้วย dynamic programming
2) จงเขียนโปรแกรมแก้ปัญหา 0/1 knapsack problem ด้วย dynamic programming
3) จงเขียน โปรแกรมแก้ปัญหา Longest Common Subsequence ด้วย dynamic programming
4) จงเขียนโปรแกรมแก้ปัญหา coin-in-a-line ด้วย dynamic programming