Algorithm Design and Analysis

วิชาบังคับก่อน: 204251 หรือ 204252; และ 206183 หรือ 206281

ผู้สอน: ตอน 1 อ. เบญจมาศ ปัญญางาม

ตอน 2 อ. ดร. จักริน ชวชาติ

บทที่ 9

Dynamic Programming Part I

บทที่ 9

Dynamic Programming

Algorithmic Paradigms

- Greedy
  - จะค่อยๆ สร้างคำตอบขึ้นมา โดยจะเลือกทำสิ่งที่ดีที่สุดในแต่ละรอบของการตัดสินใจเพื่อเลือกคำตอบ
- Divide and Conquer
  - จะแบ่งปัญหาออกเป็นปัญหาย่อย โดยแบ่งไปเรื่อยๆ จนเป็นปัญหาที่ง่ายแล้วแก้
  - จากนั้นค่อยๆ รวมคำตอบของปัญหาย่อยนั้นกลับขึ้นมาเป็นคำตอบของปัญหาตั้งต้น
- Dynamic Programming
  - จะแบ่งปัญหาออกเป็นลำดับของปัญหาย่อยที่ซ้ำกัน
  - คำนวณแล้วเก็บคำตอบไว้เพื่อที่จะได้ไม่ต้องคำนวณใหม่ทั้งหมด
  - จากนั้นนำคำตอบของปัญหาย่อยมาสร้างเป็นคำตอบของปัญหาตั้งต้น

Dynamic Programming

- Dynamic programming เป็นวิธีการแก้ปัญหาแบบหนึ่งที่
  - เราจะแตกปัญหาออกเป็นปัญหาย่อยๆ
  - จากนั้นจะเก็บผลลัพธ์ของปัญหาย่อยเหล่านี้ เพื่อที่เมื่อมีการหาคำตอบของปัญหาย่อยเหล่านี้อีกจะได้ไม่ต้องคำนวณใหม่
  - เราจึงจะได้ยินบ่อยๆ ว่า Dynamic programming กับตาราง

Dynamic Programming

- ทั้งนี้คุณสมบัติหลักของปัญหาที่จะแก้ด้วย Dynamic programming ได้คือ
  - Overlapping Subproblems (มีปัญหาย่อยซ้ำกัน เรียกให้คำนวณอันเดิมบ่อยๆ)
  - Optimal Substructure (คำตอบที่ดีที่สุดของปัญหาได้จากการใช้คำตอบที่ดีที่สุดของส่วนย่อยของปัญหา)

Overlapping Subproblems

- เช่นเดียวกับ Divide and conquer, Dynamic programming นั้นจะรวมเอาคำตอบมากจากปัญหาย่อย

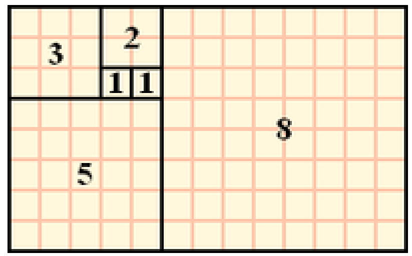
- Dynamic programming จะถูกใช้หลักๆ เมื่อคำตอบของปัญหาย่อยที่เหมือนกันนั้นถูก**คำนวณบ่อยๆ** ซึ่งใน Dynamic programming คำตอบที่คำนวณแล้วของปัญหาย่อยจะถูกเก็บไว้ในตารางเพื่อที่ว่าจะได้ไม่ต้องคำนวณใหม่อีก

- ดังนั้น Dynamic programming จะไม่เกิดประโยชน์เท่าไร เมื่อไม่มีปัญหาย่อยที่ซ้ำกันเลย (no common overlapping subproblems)

  เพราะว่าไม่รู้ว่าจะเก็บใส่ตารางไปทำไม เมื่อไม่ได้ใช้อีก

Fibonacci number

- Fibonacci number เป็นตัวอย่างของปัญหาที่มีการเรียกหาคำตอบของปัญหาย่อยซ้ำๆ กันมากๆ



- จำนวนฟีโบนัชชี หรือ เลขฟีโบนัชชี คือจำนวนต่าง ๆ ที่อยู่ในลำดับจำนวนเต็มดังต่อไปนี้ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765

Fibonacci number

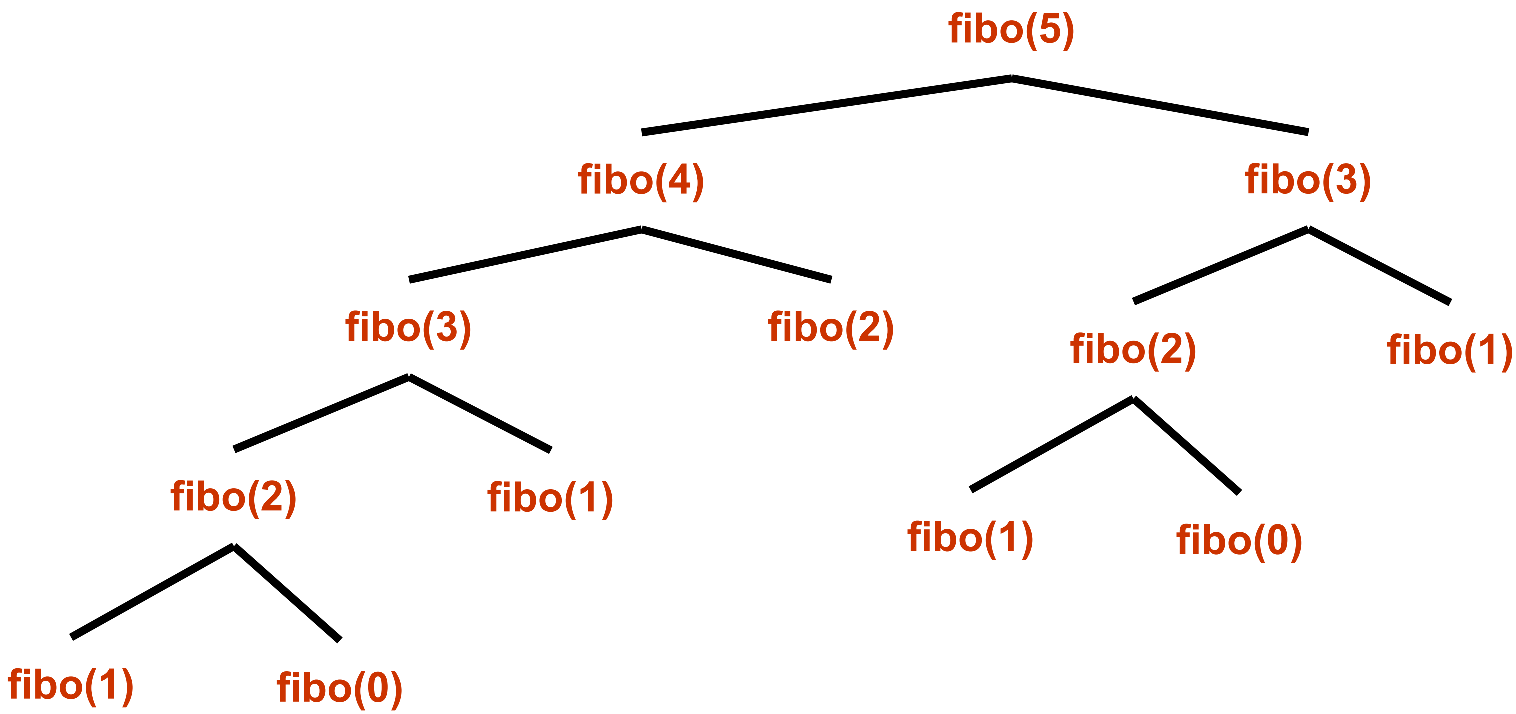
- Fibonacci number เขียนได้ในรูป

$$F_n = F_{n-1} + F_{n-2}$$

โดยที่ $F_0 = 0, F_1 = 1$

เมื่อเขียนเป็น recursive จะได้

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



- เห็นได้ว่า fibo(3) ถูกเรียก 2 ครั้ง ถ้าเราเก็บค่าของ fibo(3) แทนที่จะคำนวณใหม่อีกรอบ เราก็เอาค่าที่เก็บไว้มาใช้ได้เลย

- วิธีในการเก็บค่าเพื่อนำมาใช้ใหม่มีหลักๆ 2 วิธี
  - **Memoization** (Top down)
  - **Tabulation** (Bottom up)

Memoization (Top down)

- การจำคำตอบในลักษณะนี้**คล้ายกับการเขียนแบบ Recursive** ที่มีการปรับปรุงเล็กน้อย
- **วิธีนี้จะมองหาคำตอบใน Lookup Table ก่อนที่จะคำนวณคำตอบ**
- เราจะเริ่มด้วยการกำหนดค่า Lookup table ด้วยค่าเริ่มต้นเป็น NULL ก่อน
- เมื่อไรก็ตามที่เราต้องการคำตอบของปัญหาย่อย เริ่มต้นเราจะค้นหาใน Lookup table ก่อน
  - **ถ้ามี**ค่าที่คำนวณไว้แล้วเราก็จะนำมาใช้เลย
  - **แต่ถ้าไม่มี** เราก็จะคำนวณค่าแล้วเก็บไว้ใน Lookup Table เพื่อไว้ใช้ในคราวต่อไป

ตัวอย่าง Fibonacci number

```
int lookup[20];

void init(){
        int i;
        for(i=0;i<20;i++){
                lookup[i]=NULL;
        }
}
int fibo(int n){
        if(lookup[n] == NULL){
                if(n<=1){
                        lookup[n] = n;
                }else{
                        lookup[n]=fibo(n-1)+fibo(n-2);
                }
        }
        return lookup[n];
}
```

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

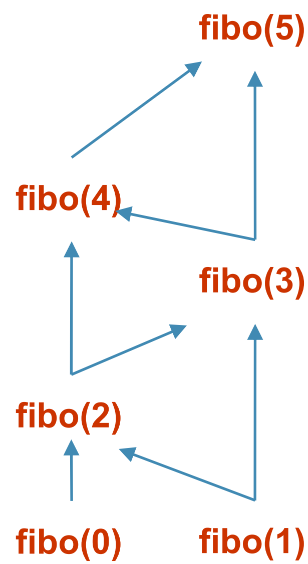
Tabulation (Bottom up)

- การสร้างตารางนั้นจะสร้างจากล่างขึ้นบน (จากตัวแรกไปตัวท้าย จากส่วนเล็กสุดสร้างคำตอบขึ้นให้ส่วนบนสุด) จากนั้นจะคืนค่าในช่องสุดท้ายจากตารางเป็นคำตอบ

- หากใช้ในตัวอย่าง Fibonacci number เราก็จะสร้าง fibo(0) จากนั้น fibo(1) จากนั้น fibo(2) จากนั้น fibo(3) ไปเรื่อยๆ

```
int fibo(int n){
    int f[n+1];
    int i;
    f[0] = 0;
    f[1] = 1;
    for(i = 2;i<=n;i++){
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

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



- ทั้งแบบ Tabulation และ Memoization นั้นจะเก็บคำตอบของปัญหาย่อย
  - ในแบบ Memoization นั้นตารางจะถูกเติมตามคำสั่งคือทำงานเมื่อถูกเรียกให้สร้างคำตอบเท่านั้น
  - ขณะที่แบบ Tabulation เริ่มจากช่องแรก จากนั้นทุกช่องจะค่อยๆ ถูกเติมลงไป
  - นั่นคือแบบ Memoization ทุกช่องใน Lookup table อาจจะไม่ถูกใส่ข้อมูล
- ทั้งนี้หากลอง Recursive เทียบกับ Tabulation และ Memoization จะพบว่าใช้เวลามากกว่ามากๆๆ (ลองหา Fibonacci number ค่ามากๆ ได้)

เปรียบเทียบ Top-Down กับ Bottom-Up

Top-Down	Bottom-Up
ข้อดี - เป็นการเปลี่ยนมาจาก Complete search แบบ Recursion - คำนวณปัญหาย่อยเมื่อจำเป็น (บางครั้งจึงเร็วกว่า)	**ข้อดี** - เร็วกว่าถ้า sub-problem ถูกเรียกบ่อยๆ เพราะว่าไม่มี overhead จาก recursive call - ประหยัดหน่วยความจำ
ข้อเสีย - ช้ากว่าถ้า sub-problem ถูกเรียกบ่อยๆ เพราะว่า overhead ของ function call (ส่วนใหญ่ในการแข่งเขียนโปรแกรมก็ผ่านอยู่ดี)	**ข้อเสีย** - เขียนยากกว่าเพราะว่าไม่ได้แปลงจาก recursive ตรงๆ

การแก้ Dynamic programming

- จริงๆ แล้วมีหลายแนวทางในการแก้
- ขั้นตอนในการแก้ DP

1. Identify if it is a DP problem
2. Decide a state expression with least parameters
3. Formulate state relationship
4. Do tabulation (or add memoization)

- How to classify a problem as a Dynamic programming problem?
  - โดยทั่วไปแล้ว ปัญหาเกี่ยวกับการ**หาค่าที่ดีที่สุด**(Optimization problem) ค่าที่มาก หรือน้อยที่สุด หรือ**ปัญหาเกี่ยวกับการนับ**ที่บอกว่าให้นับรูปแบบการจัดเรียงภายใต้เงื่อนไขบางอย่าง มักจะแก้ได้ด้วย Dynamic programming
  - ทั้งนี้ทุกปัญหา Dynamic programming นั้นจะมีคุณสมบัติ Overlapping substructure และปัญหาคลาสสิคส่วนใหญ่ของ Dynamic programming นั้นจะมี Optimal substructure เมื่อเราสังเกตได้ว่ามีคุณสมบัติสองอย่างนี้ในปัญหาจะค่อนข้างมั่นใจได้ว่าแก้ได้ด้วย Dynamic programming

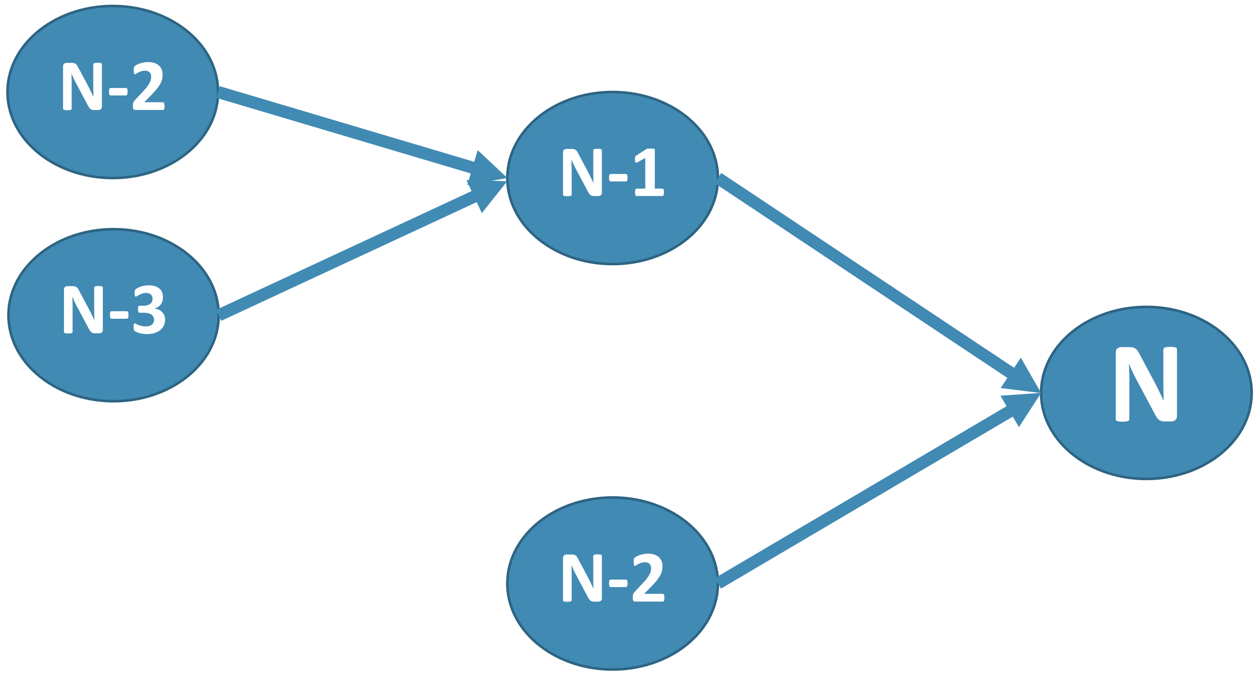
Deciding the state

- ปัญหา DP จะเกี่ยวข้องกับ State และ Transition (การเปลี่ยนสถานะ)
- state สามารถถูกนิยามได้ว่าเป็น**เซตของ parameter** ที่บ่งบอกตำแหน่งเฉพาะในปัญหานั้นได้ เซตของ parameter นี้ควรจะมีจำนวนน้อยที่สุดเท่าที่จะเป็นไปได้เพื่อที่จะเป็นการลด state space (ลดขนาดตารางนั่นเอง)

❑ Formulating a relation among the states

ความสัมพันธ์ระหว่าง state ส่วนนี้เป็นส่วนที่ยากที่สุดของการแก้ DP และต้องการความคิดสร้างสรรค์ การสังเกต และการฝึกฝน state ปัจจุบัน เกิดจาก state เก่าๆ ก่อนหน้าได้อย่างไร

Fibonacci number



1	**…**	**…**	**N-3**	**N-2**	**N-1**	**N**

Find the number of different ways to write n

Find the number of different ways to write n

ลองพิจารณาตัวอย่างต่อไปนี้

กำหนดให้ ตัวเลข 3 ตัวเลขได้แก่ 1, 3, 5

จงหาว่ารูปแบบทั้งหมดที่เป็นไปได้ในการรวมกันเป็นจำนวนที่มีค่า N จาก 3 จำนวนนี้

เช่นเลข 5

1+1+1+1+1

3+1+1

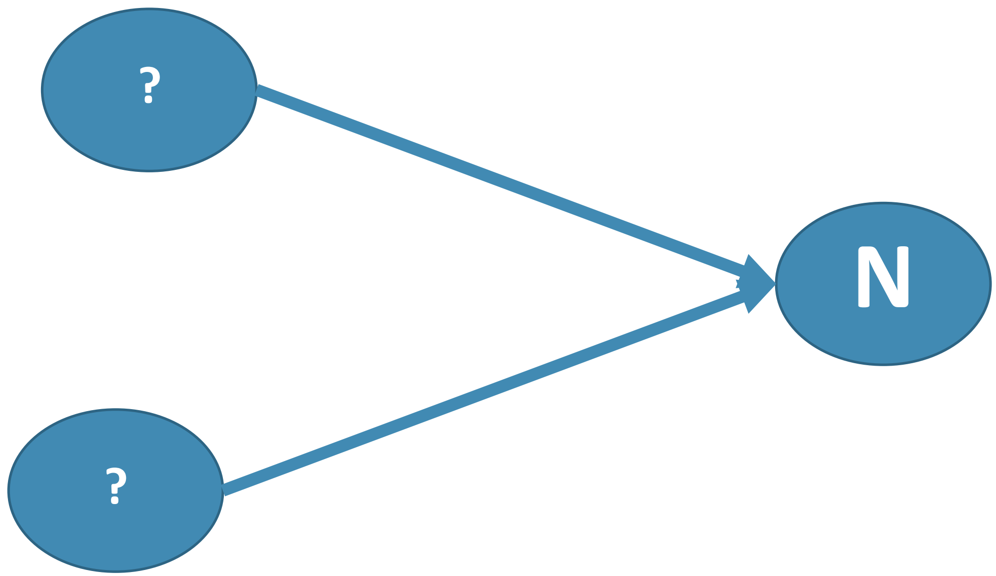
1+3+1

1+1+3

5

ลองพิจารณาปัญหาข้างต้น

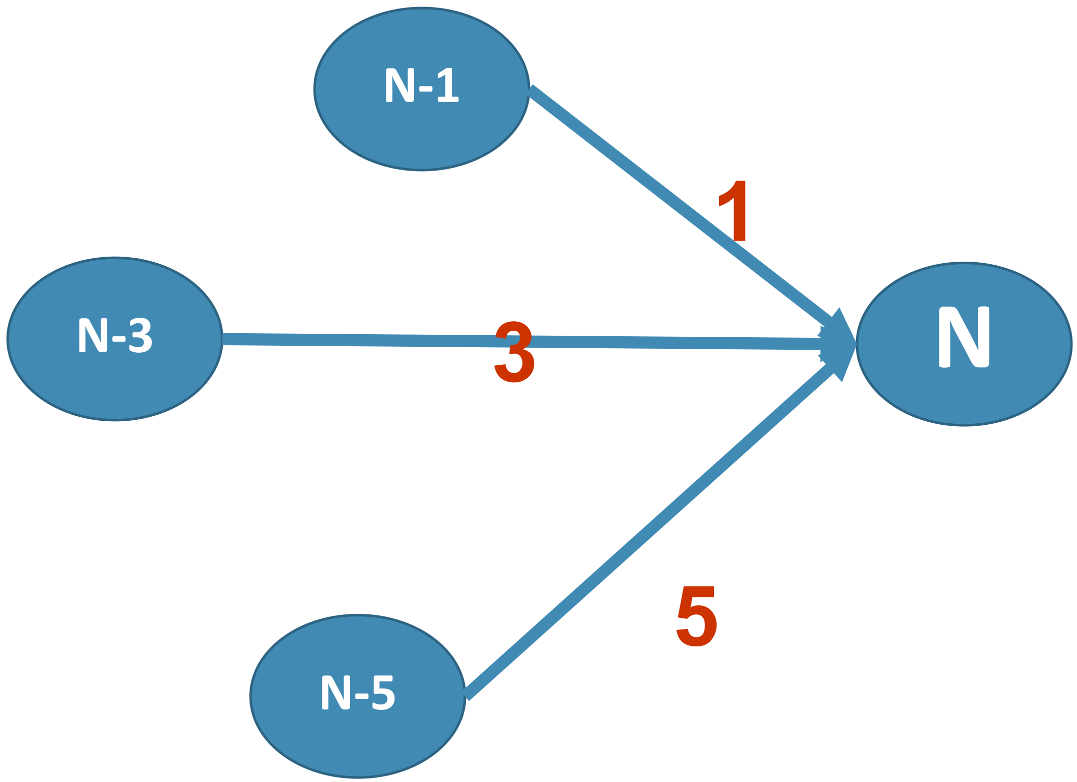
เริ่มต้นเราก็คิด "state" ของปัญหาก่อนหรือ subproblem นั่นเอง
เราจะนำเอา parameter n มาใช้ เพื่อบอกสถานะว่ามันจะระบุปัญหาย่อยใดๆ ได้อย่างไร การที่จะมาเป็นคำตอบที่ n ได้ สร้างมาได้อย่างไร



ลองพิจารณาปัญหาข้างต้น

ดังนั้น state ของ dp ก็จะหน้าตาประมาณนี้

state(n) หมายถึงจำนวนทั้งหมดในการเขียนให้ได้ค่า n โดยใช้ 1, 3, 5



- ต่อไปจะเป็นการคำนวณ state(n)
- เนื่องจากเราสามารถใช้ได้เพียง 1, 3, 5 ในการสร้าง n
- สมมติว่าเรารู้ว่าผลลัพธ์ของ n = 1, 2, 3, 4, 5, 6 นั่นคือ เรารู้ผลลัพธ์ของ state(n=1), state(n=2), state(n=3), ... , state(n=6) เมื่อเราต้องการหาค่าของ **state(n=7)** เราจะทำได้อย่างไรบ้าง
- เราจะเพิ่ม 1, 3, 5 ได้เท่านั้น นั่นคือเราสามารถรวมเป็น 7 ได้เพียง 3 วิธีคือ ใช้ 1 ต่อท้าย, ใช้ 3 ต่อท้าย, ใช้ 5 ต่อท้าย

หากใช้ 3

- หากใช้ 3 คือ เราเพิ่ม 3 ต่อท้าย
- แล้วค่าก่อนหน้าที่เราจะเพิ่มค่า 3 คืออะไร คือค่า (n=4) นั่นเอง ดังนั้น หากเราเพิ่ม 3 ต่อท้ายก็จะมีจำนวนแบบเท่ากับ state(n=4) นั่นคือ เขียน 4 ได้กี่แบบนั่นเอง

[(1+1+1+1)+3]

[(3+1)+3]

[(1+3)+3]

หากใช้ 5

- หากใช้ 5 คือ เราเพิ่ม 5 ต่อท้าย
- แล้วค่าก่อนหน้าที่เราจะเพิ่มค่า 5 คืออะไร คือค่า (n=2) นั่นเอง ดังนั้น หากเราเพิ่ม 5 ต่อท้ายก็จะมีจำนวนแบบเท่ากับ state(n=2) นั่นคือ เขียน 2 ได้กี่แบบนั่นเอง

[(1+1)+5]

หากใช้ 1

- หากใช้ 1 คือ เราเพิ่ม 1 ต่อท้าย แล้วค่าก่อนหน้าที่เราจะเพิ่มค่า 1 คืออะไร คือค่า (n=6) นั่นเอง ดังนั้น หากเราเพิ่ม 1 ต่อท้ายก็จะมีจำนวนแบบเท่ากับ state(n=6) นั่นคือ เขียน 6 ได้กี่แบบนั่นเอง
- [(1+1+1+1+1+1)+1]
- [(1+1+1+3)+1]
- [(1+1+3+1)+1]
- [(1+3+1+1)+1]
- [(3+1+1+1)+1]
- [(3+3)+1]
- [(1+5)+1]
- [(5+1)+1]

- ต่อไปลองคิดว่าทั้งสามกรณีก่อนหน้านี้ครบทุกกรณีแล้วหรือยังในการรวมกันให้ได้ 7
- ดังนั้นเราสามารถบอกว่าผลลัพธ์ของ

state(7) = state(7-1) + state(7-3) + state(7-5)

ในรูปแบบทั่วไปคือ

state(n) = state(n-1) + state(n-3) + state(n-5)

Base case

- แล้ว case ง่าย
- เราย้อนกลับมาเรื่อยๆ คำถามคือ ย้อนกลับมาถึงเมื่อไร
- ทำให้สิ่งสำคัญของ DP คือการใช้ recursive ด้วย
- ถ้าเป็น 0 ตอบ 1 วิธี
- ถ้าติดลบ เป็น 0 วิธี

```
int solve(int n)
{
    // base case
    if (n < 0)
        return 0;
    if (n == 0)
        return 1;

    return solve(n-1) + solve(n-3) + solve(n-5);
}
```

จากตัวอย่างข้างบนจะใช้เวลาในการทำงานนานมากเป็น exponential time

ดังนั้นต่อไปเราจะมาเพิ่มส่วน memoization

Adding memorization or tabulation for the state

- นี่เป็น**ส่วนที่ง่ายสุดของ** dp เราเพียงแค่เก็บ state ของคำตอบ เพื่อที่ว่าในการสอบถามคราวหน้า เราจะเอาคำตอบมาใช้ได้เลย โดยการเพิ่ม code

```
int dp[MAXN];
int solve(int n)
{
  // base case
  if (n < 0)
      return 0;
  if (n == 0)
      return 1;
  // checking if already calculated
  if (dp[n]!=0)
      return dp[n];
  // storing the result and returning
  return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);
}
```

การเปลี่ยนมาเป็น Bottom up

- Bottom up คือสร้างจากตัวเล็กประกอบมาเป็นตัวใหญ่
- ตัวเล็กคืออะไร
  - Base Case ใน recursive
  - เราก็ดูว่านอกจาก Base case ใน recursive แล้วกรณีไหนที่เราจะตอบได้เลยก็กำหนดค่าไปเลย
  - ส่วนใหญ่จะกำหนดค่าจนให้เรียก recursive ได้ทุกอัน
- ตัวใหญ่คืออะไร
  - Recursive call

```
if (n < 0)
    return 0;
if (n == 0)
    return 1;
```

แสดงว่ากรณี 0 และน้อยกว่า 0 กำหนดค่าได้เลย แต่เราสร้างจากตัวเล็ก ตัวเล็กสุดของเราคือเป็นไปได้อะไร

```
dp[0] = 1;
dp[1] = 1;
dp[2] = 1;
dp[3] = 2;
dp[4] = 3;
```

```
return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);
```

- เมื่อเราใส่ base case ครบ เราก็สามารถเติมช่องได้ทุกช่องโดยใช้ข้อมูลก่อนหน้าที่ใส่ไปแล้วได้
- เราก็ for ถัดจาก base case เลย

```
for(int i=5;i<=n;i++)
    dp[i] = dp[i-1] + dp[i-3] + dp[i-5];
```

```
int dp[MAXN];
int solve(int n)
{
    // base case
   dp[0] = 1;
   dp[1] = 1;
   dp[2] = 1;
   dp[3] = 2;
   dp[4] = 3;
   for(int i=5;i<=n;i++)
       dp[i] = dp[i-1] + dp[i-3] + dp[i-5];
   return dp[n];
}
```