Dynamic Programming

Dynamic Programming วิธีนี้จะมีการได้มาซึ่งคำตอบคล้ายๆกับ divide & conquer คือ มีแนวคิดในการแก้ปัญหาใหญ่ ด้วยการรวมคำตอบของปัญหาย่อยๆ แต่ dynamic programming จะมีวิธีลดการประมวลผลย่อยๆที่มีโอกาสจะแก้ปัญหาซ้ำซ้อนกันมากๆได้อย่างมีประสิทธิภาพ และ dynamic programming จะเหมาะมากๆกับการแก้ปัญหาแบบต้องการคำตอบที่ดีที่สุดอีกด้วย จะกล่าวว่า dynamic programming เป็นวิธีการ implementation(แปลว่า การทำให้เกิดผลหมายถึง การลงมือทำเพื่อให้โครงการหรืองานอย่างใดอย่างหนึ่งประสบผลสำเร็จ อาจใช้อุปกรณ์หรือเครื่องมือบางชนิดช่วยก็ได้) ของวิธี divide & conquer ก็ได้ ซึ่งผลที่ได้นี้จะทำให้อัตราการเจริญเติบโตของฟังก์ชั่นลดลงอย่างมากจนเป็นที่น่าพอใจ

จะต้องใช้ Dynamic Programming เมื่อใด?

Note

การจะใช้ dynamic programming เราจะต้องทราบ recurrence relation ของปัญหาก่อน

ปัญหาที่เหมาะแก่การใช้ dynamic programming คือปัญหาที่พบเหตุการณ์ดังนี้

- *Optimal sub-problem* ปัญหาต้องแก้ด้วย divide & conquer ได้ และคำตอบเกิดจากคำตอบของปัญหาย่อยที่**ดีที่สุด** เช่น MSS, closest pair เป็นต้น
- *Overlapping sub-problem* ปัญหาที่เราแบ่งย่อยมีโอกาสซ้ำกันมากๆ เมื่อใช้ dynamic programming จะทำงานได้เร็วขึ้น โดยอาศัยการทำครั้งเดียว แล้วจำไว้

เพื่อให้เห็นภาพของวิธีการแก้ปัญหาแบบ dynamic programming เราลองดูตัวอย่างสักเล็กน้อย

ตัวอย่างที่ 1 Fibonacci

ปัญหานี้เป็นปัญหาที่คุ้นเคยกันดี ทุกคนคงเคยเขียนโปรแกรมนี้มาหลายครั้งแล้ว โดย Recurrence relation ของ Fibonacci คือ

$$F(N) = F(N-1) + F(N-2)$$
$$F(1) = 1 \, , F(2)=2$$

สามารถเขียน code แบบตรงไปตรงมา ได้ดังนี้

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

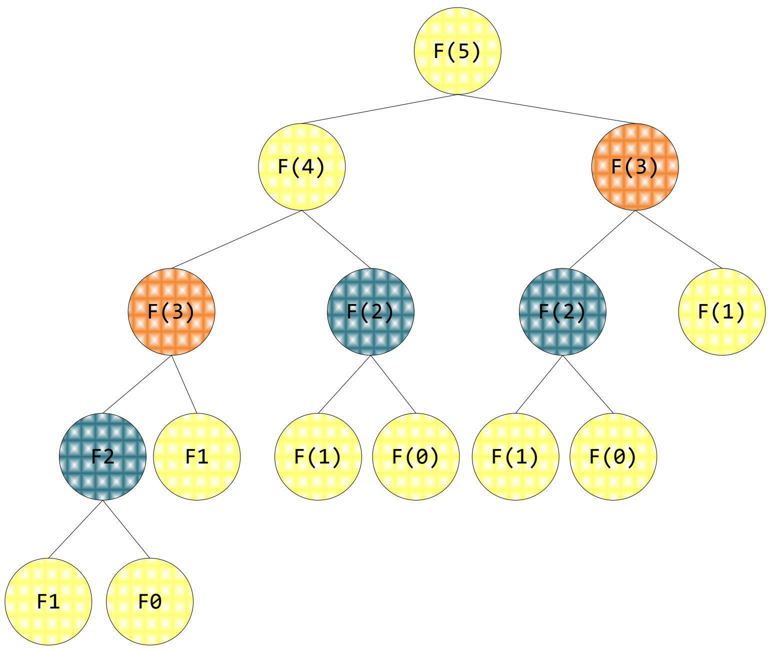
จากความรู้ทาง Discrete math ที่เคยเรียนมา ฟังก์ชั่น Fibonacci คือ

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}} = \frac{\varphi^n - (-1/\varphi)^n}{\sqrt{5}} \, ,$$

Where $\varphi = \dfrac{1+\sqrt{5}}{2} \approx 1.6180339887\ldots$

ดังนั้น เวลาการทำงานของฟังก์ชั่นนี้ คือ $\Theta(\varphi^n)$ ซึ่งเป็นฟังก์ชั่นที่โตเร็วมาก

ลองเขียน recursion tree ดูคร่าว ๆ โดยลองหา F(5) ว่าจะเรียก method ซ้ำกี่ครั้ง



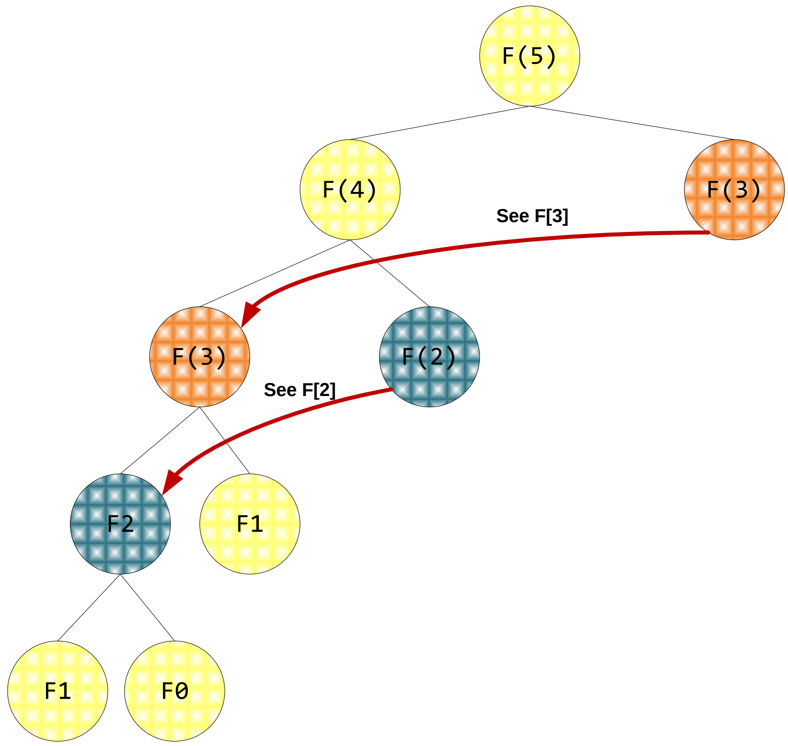
จากนั้นจเราจะพบว่า ปัญหานี้ จะเกิดการ **Overlapping sub-problem** โดยมันจะกลับไปแก้ปัญหาที่เคยแก้ไว้แล้วซ้ำหลายครั้ง (F(2) เรียกซ้ำ 3 ครั้ง F(3)เรียกซ้ำ 2 ครั้ง) ดังนั้น เราก็จะแก้ปัญหานี้ด้วยวิธี dynamic programming ได้

วิธีที่ 1 top down

ใช้วิธีการจำ(Memoization) มาประยุกต์ใช้ โดยใช้ Array เก็บผลลัพธ์ที่ทำไปแล้ว ทุกครั้งที่เรียก recursive ถ้าเคยทำมาแล้ว เอาคำตอบมาใช้ได้เลย แต่ถ้ายังไม่เคยทำ ให้ทำปกติ แล้วเก็บผลลัพธ์ลง array เมื่อทำเสร็จ สามารถเขียนโค้ดเพิ่มจากเดิมได้ดังนี้

```
int fibo(int n) {
   if (n<2) return n;
   if (F[n]>0) return F[n];
      return F[n] = fibo(n-1) + fibo(n-2);
}
```

ลองเขียน recurtion tree ดูอีกรอบหนึ่ง

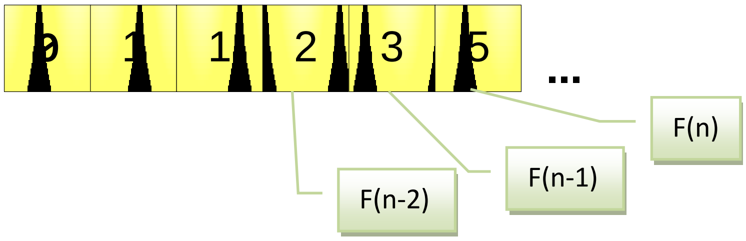


จะพบว่า จะมีการเรียก method ซ้ำ เฉพาะในกรณีที่ยังไม่เคยหาคำตอบ และจะเห็นได้ว่า ต้นไม้นี้จะไม่แตกลูกต่อในทางขวาของต้นไม้เลย เวลาการทำงานจึงขึ้นอยู่กับการเรียกซ้ำของลูกซ้ายไปจนถึงกรณีเล็กสุด ดังนั้นเวลาในการทำงานจึงเป็น $\Theta(n)$

วิธีที่ 2 bottom up

ทำง่ายๆโดยการคิดจากกรณีเล็กสุด แล้วใช้วงวน for เพื่อหาคำตอบ

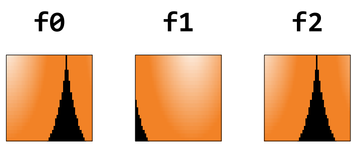
โดยกำหนด F[0] และ F[1] ให้ก่อน แล้วก็หาจาก F[2] เป็นต้นไปโดยให้ F(n) = F(n-1) + F(n-2)




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

จาก code ที่เขียนนี้ มีวงวน for เพียงอันเดียว และมีการวน n รอบ เวลาการทำงานจึงเป็น **O(n)**

แต่ จากที่เห็น ถ้าหาข้อมูล 10,000 ตัว ต้องสร้าง Array 10,000 ช่อง ซึ่งก็เป็นการเปลืองข้อมูลในหน่วยความจำมากเกินไป และจะเห็นได้ว่า ข้อมูลที่ใช้ในการหา F(n) จะใช้เพียงแค่ F(n-1) กับ F(n-2) เท่านั้น ดังนั้นเราใช้แค่เพียง 3 ช่อง ก็เพียงพอแล้ว




```
int fibo(int n){
      int f0 = 0,f1=1,f2=1
      for(int i=2;i<=n;i++){
            f2 = f0 + f1;
            f0 = f1;
            f1 = f2;
      }
      return f2;
}
```

ตัวอย่างที่ 2 Binomial Coefficient (ปัญหาการเลือกสิ่งของ)

ปัญหาการเลือกสิ่งของ เป็นปัญหาที่เรียนกันมาหลายรอบ ตั้งแต่มัธยมต้น มัธยมปลาย และวิชา Discrete math กันแล้ว ซึ่งปัญหาก็คือ มีวิธีการเลือกสิ่งของ k ชิ้น จากสิ่งของ n ชิ้น กี่วิธี โดยปัญหานี้ มีวิธีแก้ปัญหาหลายวิธี ยกตัวอย่างเช่น สมการแฟกทอเรียล ที่คุ้นเคยกันดี คือ

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \quad \text{for} \;\; 0 \le k \le n.$$

มีวิธีแก้ปัญหานี้อีกวิธีหนึ่งคือ ใช้นิยามแบบ Recursive

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k > 0,$$

$$\binom{n}{0} = 1 \quad \text{for all } n \in \mathbb{N},$$

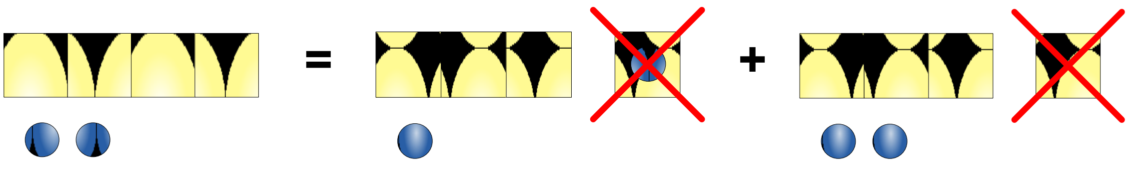
$$\binom{0}{k} = 0 \quad \text{for all integers } k > 0.$$

(ขอขอบคุณข้อมูลภาพจาก http://en.wikipedia.org/wiki/Binomial_coefficient#Recursive_formula)

จากนิยามแบบ Recursive นี้ แปลความได้ว่า จำนวนวิธีในการเลือกสิ่งของ k ชิ้น จาก n ชิ้น คือ จำนวนวิธีในการเลือกของ k-1 ชิ้นจาก n-1 ชิ้น รวมกับ จำนวนวิธีการเลือกของ k ชิ้นจาก n-1 ชิ้น

จะขอแสดงตัวอย่างจากรูปภาพเพื่อให้เข้าใจมากขึ้น โดยกำหนดปัญหาคือ จะมีวิธีการกี่วิธีในการใส่ลูกบอล k ลูกลงใน กล่อง n กล่อง โดยที่ลูกบอลทุกลูกเหมือนกัน

เช่น **C(4,2) = C(3,1) + C(3,2)**

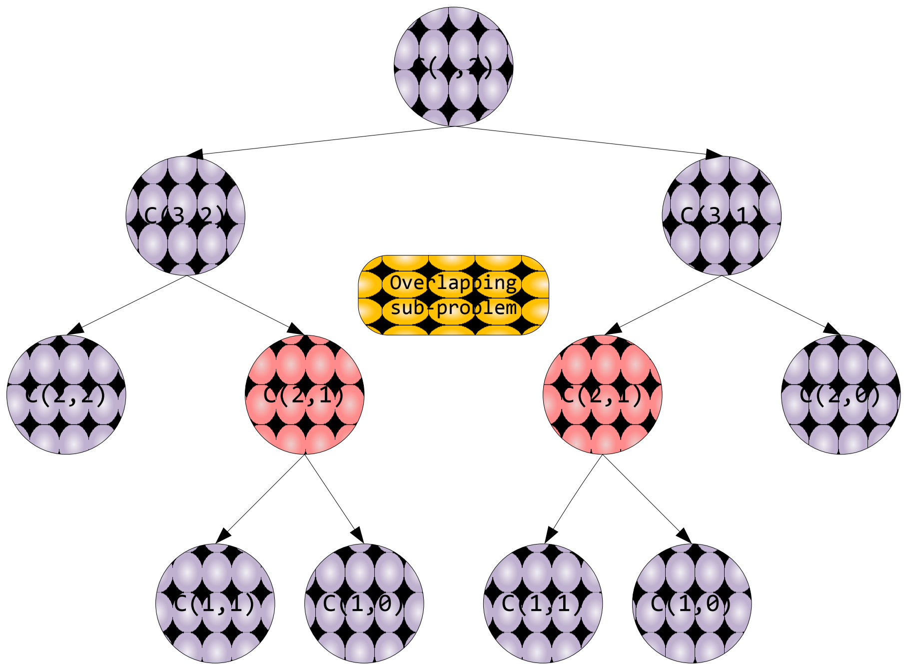


จากภาพ จะเห็นได้ว่า วิธีการเลือกใส่ของ 2 ชิ้น ใส่กล่อง 4 กล่อง สามารถลดจำนวนการวิเคราะห์ลงเหลือแค่ 3 กล่อง ได้ แต่ในที่นี้ เมื่อตัดจำนวนกล่องออกไป 1 ก็จะทำได้สองวิธีคือ กล่องนั้นถูกเลือกให้ใส่บอล หรือไม่ถูกเลือกให้ใส่บอล แล้วเอาวิธีนำบอลใส่กล่อง 3 กล่องของทั้ง 2 วิธีมารวมกัน ทำเช่นนี้ไปเรื่อยๆจนถึงกรณีเล็กสุดก็จะได้คำตอบ

จากความสัมพันธ์ดังกล่าว ปัญหา Binomial Coefficients สามารถเขียนเป็น code ได้ดังนี้

```
int bino_naive(int n,int r) {
      if (r == n) return 1;
      if (r == 0) return 1;
      int result = bino_naive(n-1,r) + bino_naive(n-1,r-1);
      return result;
}
```

ทดลองเขียน Recursion tree ดูอย่างคร่าว ๆ



จากภาพ จะเห็นได้ว่า ปัญหานี้มีการแบ่งเป็นปัญหาย่อยแล้วเจอปัญหาที่ซ้ำกัน หรือที่เรียกว่า overlapping sub-problem ซึ่งในกรณีนี้ จะใช้วิธี dynamic programming ในการแก้ปัญหาได้

วิธีที่ 1 top down

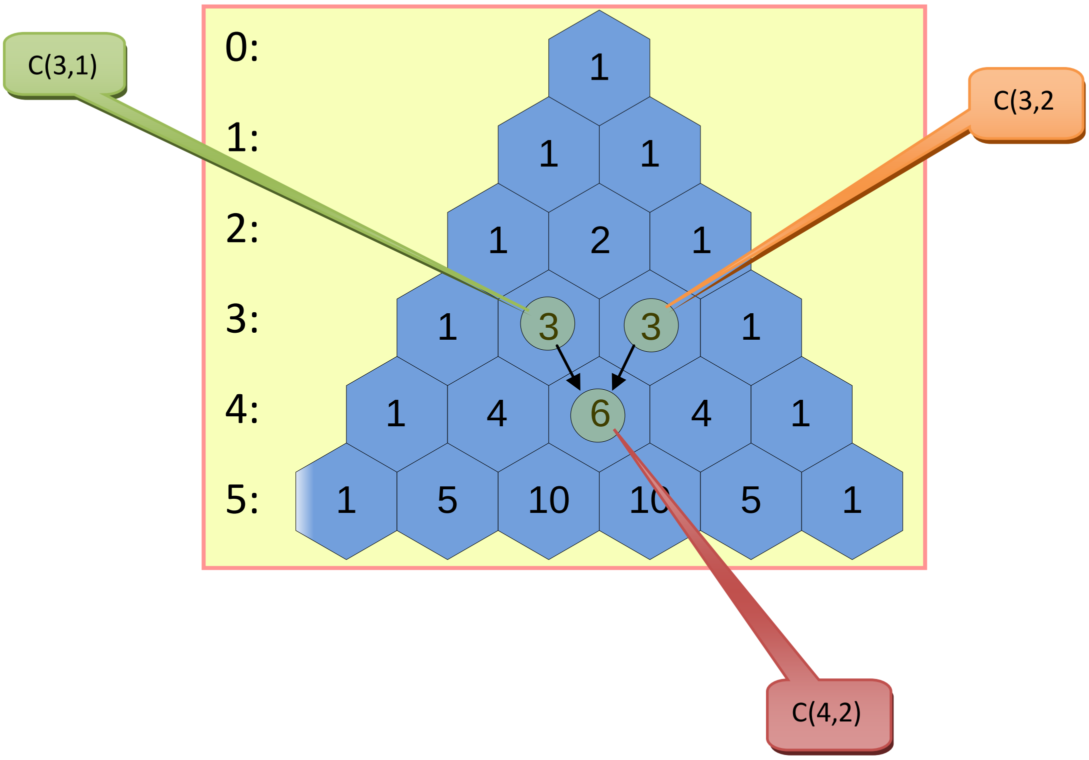
ใช้วิธีการจำ(Memoization) มาประยุกต์ใช้โดยใช้ตาราง(Array 2 มิติ)มาเก็บแล้วจำไว้ ก็จะเขียนโค้ดเพิ่มได้ดังนี้คือ

```
int bino_memoize(int n,int r) {
      if (r == n) return 1;
      if (r == 0) return 1;

      if (storage[n][r] != -1)
            return storage[n][r];

      int result = bino_memoize(n-1,r) + bino_memoize(n-1,r-1);
      storage[n][r] = result;

      return result;
}
```

การเขียน code เพิ่มในส่วนนี้ ก่อนที่โปรแกรมของเราจะไปทำปัญหาย่อย โปรแกรมจะไปตรวจสอบดูว่า สิ่งที่กำลังจะทำนั่นเคยทำแล้วหรือยัง ถ้าทำแล้วก็เอาผลคำตอบใช้ได้เลย ถ้ายังก็ไปทำแล้วเก็บผลที่ทำนั้นไว้ด้วย

วิธีที่2 bottom up

การมองเป็น bottom up คือมองจากกรณีเล็กๆเพื่อแก้ปัญหาที่ใหญ่ โดยคิดจากกรณีเล็กๆขึ้นไปเรื่อยๆ โดยการรวมขึ้นไปเรื่อยๆจะได้เป็นสามเหลี่ยมปาสคาล(Pascal's triangle)



แนวคิด - ในกรณีของสามเหลี่ยมปาสคาลนี้ เราก็พอจะเขียนโปรแกรมให้อยู่ในรูปแบบของตารางได้ดังนี้ คือ

k

n

	0	1	2	3	4	5
0	1					
1	1 +	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

แต่จากแนวคิดนี้ยังไม่เป็นที่น่าพอใจนัก คือยังมีข้อมูลส่วนที่เราไม่จำเป็นต้องไปคิด เพราะมันไม่เกี่ยวกับคำตอบที่เราจะต้องการหา ยกตัวอย่างเช่น

หากเราต้องการหา **C(5,2)** เราไม่จำเป็นต้องทำในกรณีที่ k มากกว่า 2 เลย เพราะการหาคำตอบในส่วนนั้นไม่ได้มีส่วนช่วยในการหาคำตอบที่ต้องการ

k

n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

ไม่จำเป็นต้องทำในส่วนนี้

เมื่อได้แนวคิดที่ต้องการแล้ว ก็นำมาเขียน code ได้ดังนี้

```
int bino_bottomup(int n,int k){
    int C[n+1][k+1];
    for(int i=0;i<=n;i++)C[i][0] =1; //store 1 to C[n][0]
    for(int i=0;i<=k;i++)C[i][i] =1; //store 1 to C[n][k] ;n=k

    for(int i=0;i<=n;i++)
        for(j=1;j<=k&&j<i;j++)
            C[i][j] = C[i-1][j]+C[i-1][j-1];

    return C[n][k];
}
```

เนื่องจากการทำงานของโปรแกรมนี้ มีจำนวนข้อมูลแปรตามช่องของ n และ k ดังนั้น เวลาการทำงานคือ $\Theta(nk)$

จัดทำโดย

นาย ณัฐวุฒิ สาระทัศนานันท์ 5130189421

นางสาว ชุดา กิตติพรพิมล 5130136121