Dynamic Programming

Dynamic Programming

- Dynamic programming เป็นวิธีการแก้ปัญหาแบบหนึ่งที่เราจะแตกปัญหาออกเป็นปัญหาย่อยๆ จากนั้นจะเก็บผลลัพธ์ของปัญหาย่อยเหล่านี้ เพื่อที่เมื่อมีการหาคำตอบของปัญหาย่อยเหล่านี้อีกจะได้ไม่ต้องคำนวณใหม่ เราจึงจะได้ยินบ่อยๆ ว่า Dynamic programming กับตาราง

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

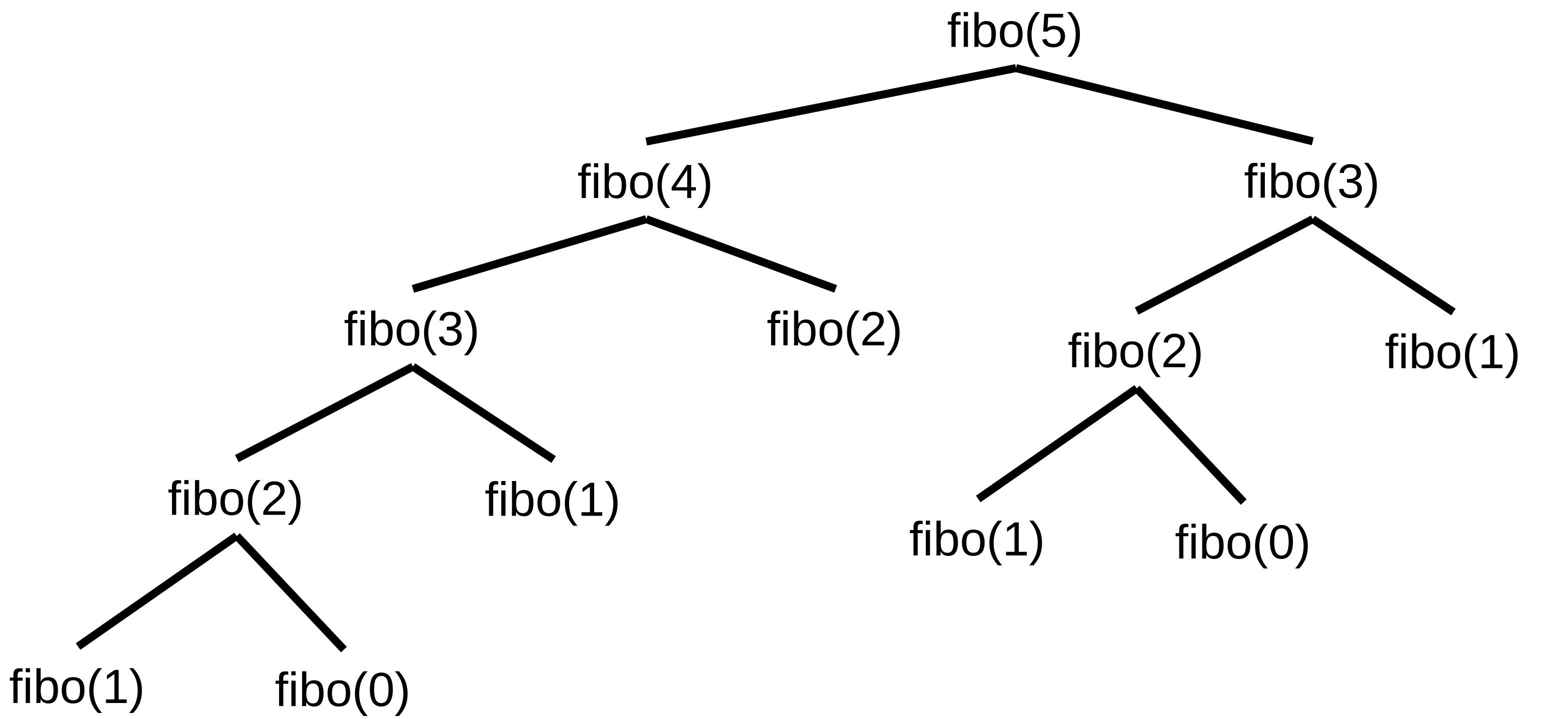
Overlapping Subproblems

- เช่นเดียวกับ Divide and conquer, Dynamic programming นั้นจะรวมเอาคำตอบมากจากปัญหาย่อย
- Dynamic programming จะถูกใช้หลักๆ เมื่อคำตอบของปัญหาย่อยที่เหมือนกันนั้นถูกคำนวณบ่อยๆ ซึ่งใน Dynamic programming คำตอบที่คำนวณแล้วของปัญหาย่อยจะถูกเก็บไว้ในตารางเพื่อที่ว่าจะได้ไม่ต้องคำนวณใหม่อีก
- ดังนั้น Dynamic programming จะไม่เกิดประโยชน์เท่าไรเมื่อไม่มีปัญหาย่อยที่ซ้ำกันเลย(no common overlapping subproblems) เพราะว่าไม่รู้ว่าจะเก็บใส่ตารางไปทำไม เมื่อไม่ได้ใช้อีก ตัวอย่างเช่น binary search ไม่มีปัญหาย่อยที่ซ้ำกัน

Fibonacci number

- Fibonacci number เป็นตัวอย่างของปัญหาที่มีการเรียกหาคำตอบของปัญหาย่อยซ้ำๆ กันมากๆ

```
int fibo(int n){
      if(n==0 || n==1)
            return 1;
      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)

- การจำคำตอบในลักษณะนี้คล้ายกับการเขียนแบบ Recursive ที่มีการปรับปรุงเล็กน้อย
- วิธีนี้จะมองหาคำตอบใน Lookup Table ก่อนที่จะคำนวณคำตอบ
- เราจะเริ่มด้วยการกำหนดค่า Lookup table ด้วยค่าเริ่มต้นเป็น NULL ก่อน เมื่อไรก็ตามที่เราต้องการคำตอบของปัญหาย่อย เริ่มต้นเราจะค้นหาใน Lookup table ก่อน ถ้ามีค่าที่คำนวณไว้แล้วเราก็จะนำมาใช้เลย แต่ถ้าไม่ใช่เราก็จะคำนวณค่าแล้วเก็บไว้ใน 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];
}
```

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];
}
```

- 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

Optimal substructure

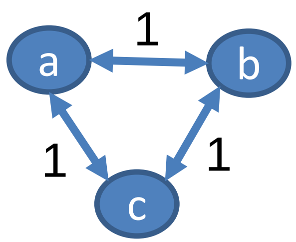
ปัญหาจะมี Optimal substructure ถ้าคำตอบที่ดีที่สุด (Optimal solution) ของปัญหานั้นสามารถหาคำตอบได้จากการใช้คำตอบที่ดีที่สุดของปัญหาย่อยของมัน

<u>ตัวอย่างเช่น</u> ปัญหา Shortest path

ถ้าโหนด x อยู่ใน shortest path จากโหนด(Source) u ไปโหนด v แล้ว shortest path จาก u ไป v คือการรวมกันของ shortest path จาก u ไป x แล้วจาก x ไป v

แต่ในทางตรงข้ามปัญหา Longest path นั้นไม่มี Optimal substructure

ตัวอย่างเช่นระยะทางยาวที่สุดจาก a ไป c นั้นคือ a->b->c แต่ระยะทางที่ยาวที่สุดจาก a ไป b คือ a->c->b



นั่นคือ ระยะทางยาวที่สุดจาก a ไป c ไม่ได้เกิดจาก ระยะทางยาวที่สุดจาก a ไป b รวมกับ ระยะทางยาวที่สุดจาก b ไป c

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)

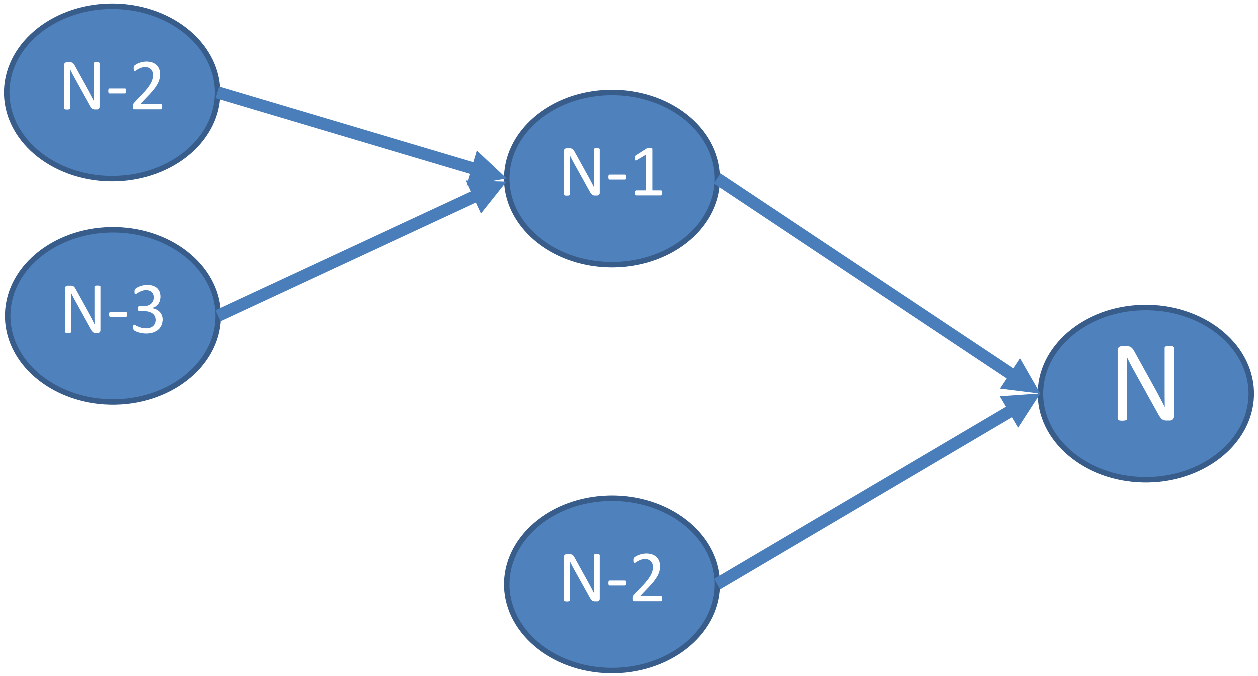
- **How to classify a problem as a Dynamic programming problem?**
  - โดยทั่วไปแล้ว ปัญหาเกี่ยวกับการหาค่าที่ดีที่สุด(Optimization problem) ค่าที่มาก หรือน้อยที่สุด หรือปัญหาเกี่ยวกับการนับที่บอกว่าให้นับรูปแบบการจัดเรียงภายใต้เงื่อนไขบางอย่าง มักจะแก้ได้ด้วย Dynamic programming
  - ทั้งนี้ทุกปัญหา Dynamic 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**

ลองพิจารณาตัวอย่างต่อไปนี้

กำหนดให้ ตัวเลข 3 ตัวเลขได้แก่ 1, 3, 5 จงหาว่ารูปแบบทั้งหมดที่เป็นไปได้ในการรวมกันเป็นจำนวนที่มีค่า N จาก 3 จำนวนนี้ เช่นเลข 5

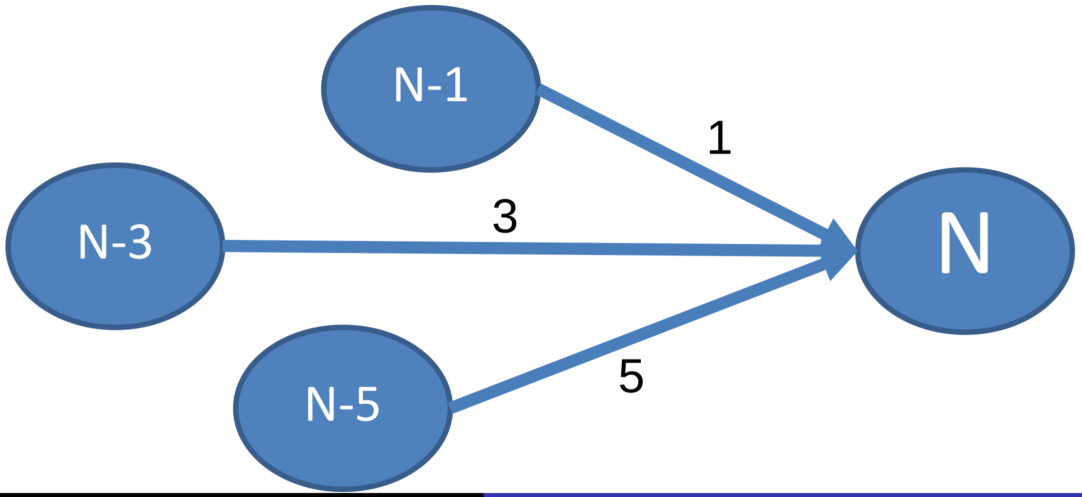
1+1+1+1+1 3+1+1 1+3+1 1+1+3 5

ลองพิจารณาปัญหาข้างต้น

เริ่มต้นเราก็คิด "state" ของปัญหาก่อน เราจะนำเอา parameter n มาใช้เพื่อบอกสถานะว่ามันจะระบุปัญหาย่อยใดๆ ได้อย่างไร

นั่นคือ สถานะที่ n, ตำแหน่งที่ n, ค่าที่ n, ชิ้นที่ n มีลักษณะอย่างไร

ดังนั้น state ของ dp ก็จะหน้าตาประมาณนี้ state(n) โดยที่ 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 วิธี คือ ใช้ 1 ต่อท้าย, ใช้ 3 ต่อท้าย, ใช้ 5 ต่อท้าย

- หากใช้ 3 คือ เราเพิ่ม 3 ต่อท้าย แล้วค่าก่อนหน้าที่เราจะเพิ่มค่า 3 คือ อะไร 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 คือ เราเพิ่ม 5 ต่อท้าย แล้วค่าก่อนหน้าที่เราจะเพิ่มค่า 5 คือ อะไร คือค่า (n=2) นั่นเอง ดังนั้น หากเราเพิ่ม 5 ต่อท้ายก็จะมีจำนวนแบบเท่ากับ state(n=2) นั่นคือ เขียน 2 ได้กี่แบบนั่นเอง

[(1+1)+5]

- หากใช้ 1 คือ เราเพิ่ม 1 ต่อท้าย แล้วค่าก่อนหน้าที่เราจะเพิ่มค่า 1 คือ อะไร คือค่า (n=6) นั่นเอง ดังนั้น หากเราเพิ่ม 1 ต่อท้ายก็จะมีจำนวน แบบเท่ากับ 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)

- 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
- นี่เป็นส่วนที่ง่ายสุดของ 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);
}
```

```
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];
}
```

แบบฝึกหัด

Cutting a rod กำหนดเชือกความยาว n หน่วยและตารางราคา(ทุกแบบที่น้อยกว่า n)ที่บอกว่าเชือกว่ายาว x หน่วยราคากี่บาท ตัวอย่างเช่น

ความยาว	1	2	3	4	5	6	7	8
ราคา	1	5	8	9	10	17	17	20

พบว่า ถ้าเชือกยาว 8 หน่วย ราคาดีที่สุดคือ 22

ความยาว	1	2	3	4	5	6	7	8
ราคา	3	5	8	9	10	17	17	20

ตัวอย่างนี้พบว่า ถ้าเชือกยาว 8 หน่วย ราคาดีที่สุดคือ 24

ให้ลองหา state และความสัมพันธ์

ปัญหา Classic ของ DP

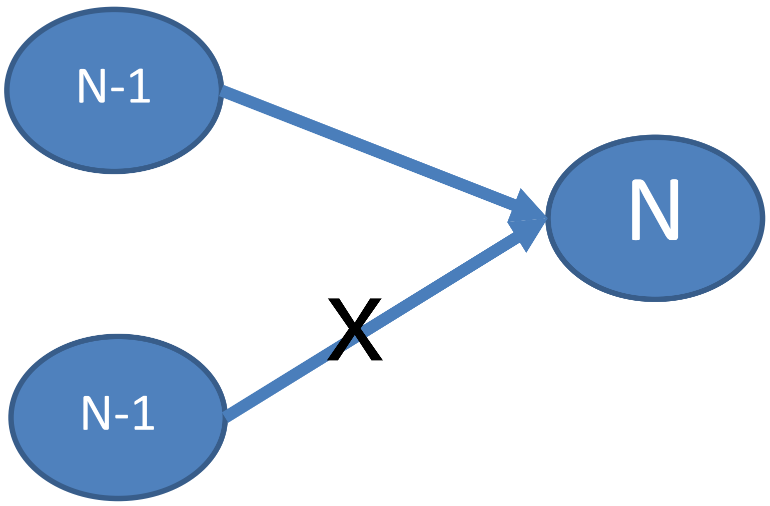
- จริงๆ แล้วปัญหา DP มีมากมาย แต่มี 6 classic DP problem ที่ควรคล่องเลย จากนั้นพอได้พื้นฐานจาก 6 อันนี้แล้วก็ลองดูพวก non-classic
  - Max 1D Range Sum
  - Max 2D Range Sum
  - Longest Increasing Subsequence(LIS)
  - 0-1 Knapsack (Subset Sum)
  - Coin Change
  - Traveling Salesman Problem (TSP)

Max 1D Range Sum

- Max 1D Range Sum หรือ Maximum Contiguous Subsequence Sum หรือ Largest Sum Contiguous Subarray หรือ Maximum Sum Contiguous Subsequence

- คือมีลำดับของตัวเลข จงหาผมรวมของตัวที่ติดกันที่มากที่สุด
- เช่น 5 -2 3 ตอบ 6 คือรวมตั้งแต่ตัวแรกถึงตัวสุดท้าย
- เช่น 5 -6 3 6 ตอบ 9 คือเอาแค่สองตัวสุดท้าย

- ลองหา state และ transition มองว่าตัวสุดท้ายเกิดอะไรขึ้น


N-1
N-1
N
X

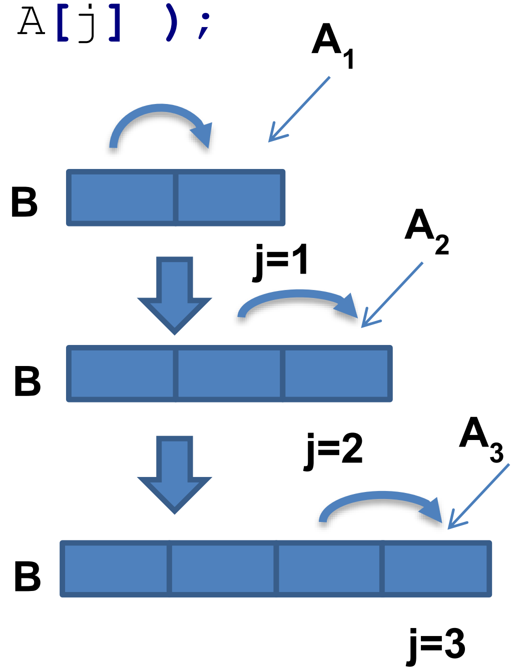
- $a_0, a_1, \ldots a_{n-1}$
- $S(0) = a_0$
- $S(1) = \max(a_0+a_1, a_1)$
- …
- $S(j) = \max(S(j-1)+a_j, a_j)$

- ข้อสังเกต S(j) เป็นค่ามากสุดที่ตำแหน่ง j ไม่ใช่มากสุดของทั้งหมดนะ

Recursive version

```
#include<bits/stdc++.h>
using namespace std;
int g_max;
int a[3]={5,-2,3};
int MCSS(int n){
    if(n==0){
        g_max = a[0];
        return a[0];
    }else{
        int temp = max(MCSS(n-1)+a[n],a[n]);
        g_max = max(g_max,temp);
        return temp;
  }
}
int main(){
    MCSS(2);
    cout<<g_max;
    return 0;
}
```

```
int maxContiguousSum(int A[], int len) {
        int j;
        int B[len];
        B[0] = A[0];
        for (j = 1; j < len; j++) {
                B[j] = max(B[j-1]+A[j], A[j] );
        }
        int max_so_far = B[0];
        for (j = 1; j < len; j++) {
                if( max_so_far < B[j])
                max_so_far = B[j];
        }
        return max_so_far;
}
```



```
int maxContiguousSum(int A[], int n){
        int j;
        int max_so_far = A[0], i;
        int curr_max = A[0];
        for (j = 1; j < n; j++){
          curr_max = max(curr_max + A[j] , A[j]);
          max_so_far = max(curr_max, max_so_far );
        }
        return max_so_far;
}
```

Longest Increasing Subsequence

- กำหนดลำดับ {A[0],A[1],...A[n−1]} มาให้ ให้หาลำดับย่อยที่ไม่จำเป็นต้องติดกันที่มีค่าเพิ่มขึ้นที่ยาวที่สุด (คำว่า Subsequence แสดงว่าไม่จำเป็นต้องต่อเนื่อง)
- ตัวอย่างเช่น n = 8 A={−7,10,9,2,3,8,8,1}
- จะได้ว่า {−7,2,3,8} เป็นลำดับย่อยที่เพิ่มขึ้นที่ยาวที่สุด