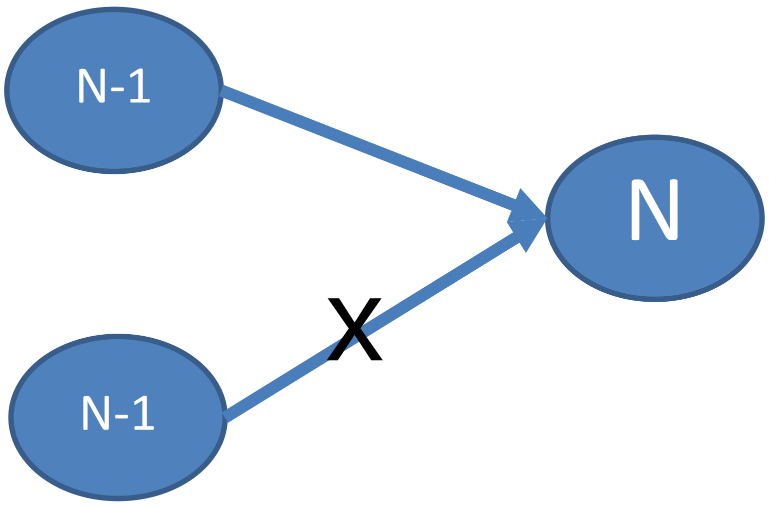
Dynamic Programming

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

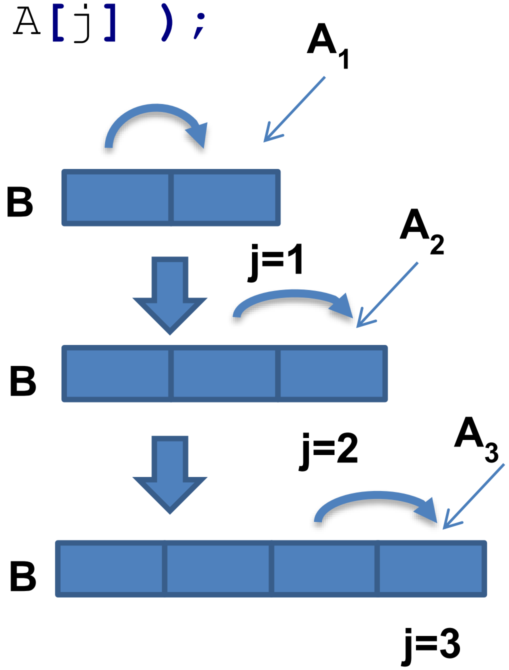
- a_0, a_1, ...a_{n-1}
- $S(0) = a_0$
- $S(1) = \max(a_0+a_1, a_1)$
- ...
- $S(j) = \max(S(j-1)+a_j, a_j)$

- 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;
}
```

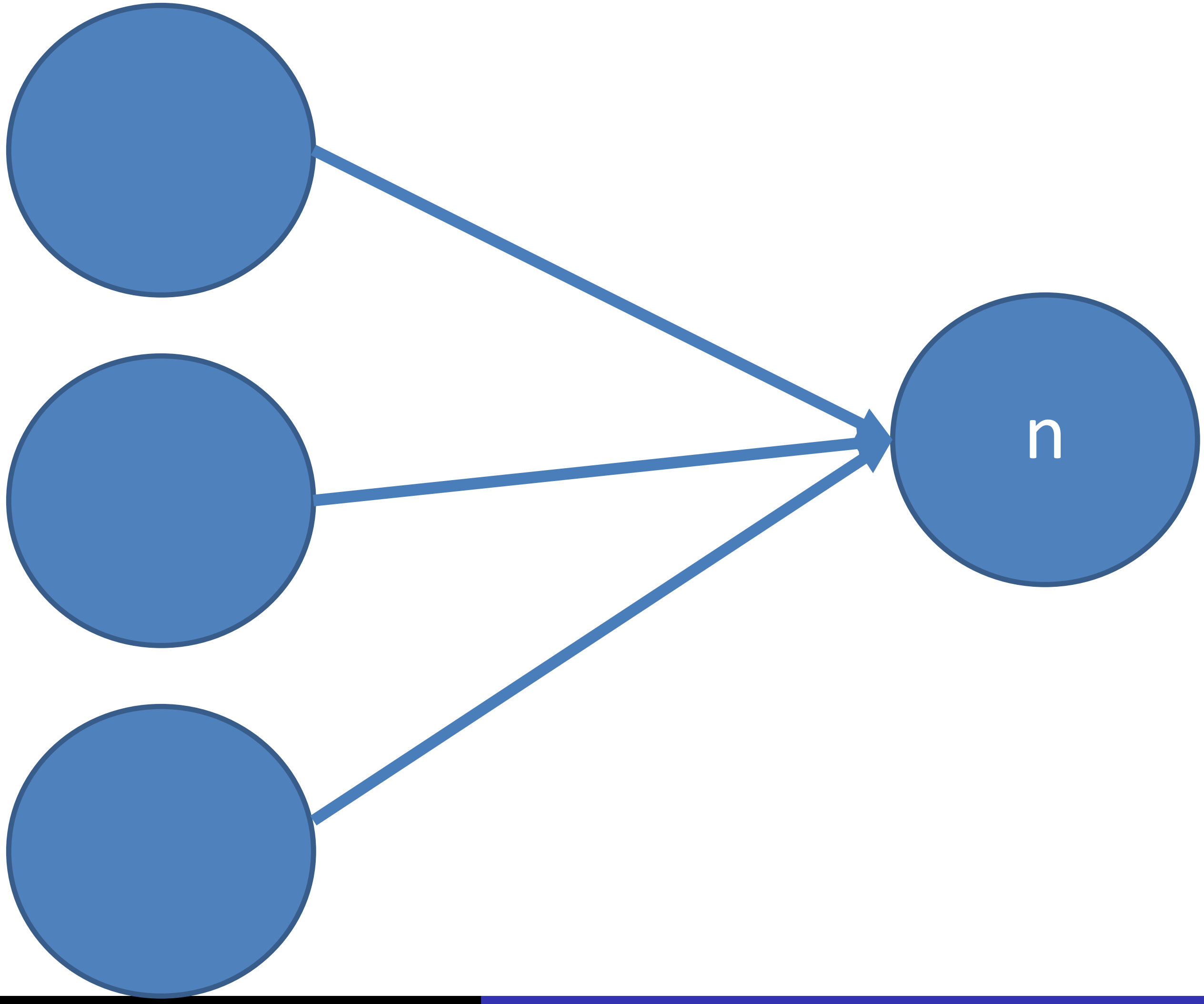
ปัญหา Classic ของ DP

- 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)

Coin Change

- ปัญหานี้ กำหนดจำนวนเงินที่ต้องการมาให้ V และ list ของเหรียญ n เหรียญ มาให้นั่นคือเราจะได้ coinValue[i] สำหรับเหรียญชนิดที่ i∈ [0,...,n−1] แล้วคำถามคือจำนวนเหรียญที่น้อยที่สุดที่รวมกันได้ V เป็นเท่าไร สมมติว่าเรามีเหรียญย่อยไม่จำกัด
- ตัวอย่างเช่น V= 10 n=2, coinValue={1,5} เราแลกได้คือ
  - ใช้ 1 บาท 10 เหรียญ
  - ใช้ 5 บาท 1 เหรียญและ 1 บาท 5 เหรียญรวมเป็น 6 เหรียญ
  - ใช้ 5 บาท 2 เหรียญ

- เราสามารถใช้ Greedy ถ้าเหรียญนั้นออกแบบมาดี แต่ในกรณี๊ทั่วไปเราต้องใช้ DP
- ตัวอย่างเช่น V = 7 n = 4, cointValue={1,3,4,5} ได้เท่าไร


n

- แนวทางคือใช้ complete search recurrence บนความสัมพันธ์ของ change(value) เมื่อ value คือปริมาณที่เหลือของเหรียญที่เราจะแลก ดังนั้น

- Change(0) = 0
- Change(<0) = inf
- Change(value) – 1+ min (change(value–coinValue[i])) for all I in [0,n–1]
- จากนั้นคำตอบคือการ return ค่า change(V)

- สมมติว่ามีเหรียญ {1,3,4,5}

<0	0	1	2	3	4	5	6	7	8	9
inf										

0–1 Knapsack (Subset Sum)

- ปัญหานี้ กำหนดสิ่งของมาให้ n ชิ้น แต่ละชิ้นมีมูลค่า V_i และมีน้ำหนัก W_i สำหรับทุก $i \in [0..n-1]$ และให้ถุงเป้ที่มีความจุขนาด S หน่วย จงคำนวณมูลค่าของสิ่งของมากสุดที่เราสามารถบรรจุลงในถุงเป้ได้ โดยเราสามารถเลือกหรือไม่เลือกสิ่งของ (แบ่งเป็นชิ้นย่อยๆ ไม่ได้ จึงเรียกว่า 0–1 นั่นคือ เลือกหรือไม่เลือก)

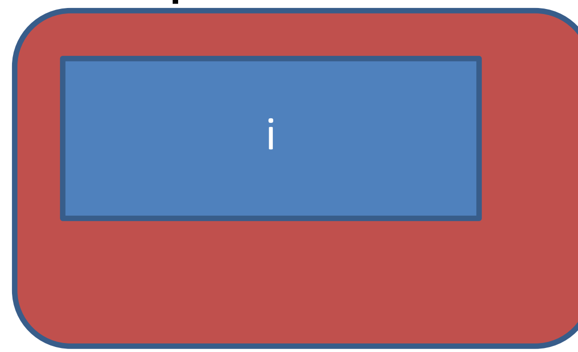
- ปัญหานี้รู้จักในอีกชื่อหนึ่งว่า Subset Sum Problem นั่นคือ มีเซตของจำนวนเต็มมาให้และมีเลขจำนวนเต็ม S มี subset ที่ไม่ว่างที่รวมกันได้ S หรือไม่
- ถ้าสิ่งของแบ่งแยกย่อยได้ เรียกว่า Fractional Knapsack

- ตัวอย่างเช่น มีของ 4 ชิ้น val={100,70,50,10}, wt={10,4,6,12}, W=12

- ถ้าเราเลือกของชิ้นที่ 0 มีน้ำหนัก 10 มีค่า 100 เราจะหยิบของอื่นอีกไม่ได้ ไม่ใช้คำตอบดีสุด
- ถ้าเราเลือกของชิ้นที่ 3 มีน้ำหนัก 12 มีค่า 10 เราจะหยิบของอื่นอีกไม่ได้ ไม่ใช้คำตอบดีสุด
- ถ้าเราเลือกของชิ้นที่ 1 และ 2 มีน้ำหนักรวม 10 มีค่า 120 เราจะได้ค่ามากสุด

- วิธีแก้แบบถึก
- เราก็พิจารณาทุก subset ที่เป็นไปได้จากนั้นคำนวณน้ำหนักทั้งหมดของสิ่งของใน subset
- จากนั้นก็พิจารณา subset ที่น้ำหนักไม่เกิน W
- แล้วเลือกเอาอันมากสุด

- เมื่อพิจารณาของชิ้นที่ i เราทำอะไรได้บ้าง
- เราเลือก/ไม่เลือก
- ถ้าเราเลือก นั่นหมายความว่า มีของชิ้นที่ i ในถุงเป้แล้ว
  - ถุงเป้เหลือความจุเท่าไร
  - แล้วเราไปถามค่าที่ดีที่สุดของใครต่อ



- ถุงเป้เหลือความจุ $W-wt_i$ คือความจุปัจจุบัน– น้ำหนักของชิ้นที่i
- แล้วค่าที่ดีที่สุดที่เหลือคือ พิจารณาของ i–1 ชิ้น โดยที่ถุงมีความจุแค่ $W-wt_i$ นั่นเอง

- ถ้าเราไม่เลือก นั่นหมายความว่า ไม่มีของชิ้นที่ i ในถุงเป้แน่ๆ
  - ถุงเป้เหลือความจุเท่าไร
  - แล้วเราไปถามค่าที่ดีที่สุดของใครต่อ
- ถุงเป้เหลือความจุเท่าเดิม
- ของที่พิจารณาก็มีแค่ i-1 ชิ้น
- ค่าที่ดีที่สุดคือ พิจารณาของ i-1 ชิ้น โดยที่ถุงมีความจุเดิมหรือ W นั่นเอง

- **วิธีแก้**

```
int wt[]
int val[]
int knapSack(int W, int n){// Base Case
   if (n == 0 || W == 0)
       return 0;
   //  ถ้าของใหญ่กว่าถุงเป้ ไม่รวม
   if (wt[n-1] > W)
       return knapSack(W, n-1);
   //  สิ่งที่เกิดขึ้นได้มี  2 อย่าง:
   // (1) nth item included
   // (2) not included
   else return max( val[n-1] + knapSack(W-wt[n-1], n-1),
                    knapSack(W, n-1)
       );
}
```

```
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n+1][W+1];
    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++){
        for (w = 0; w <= W; w++){
            if (i==0 || w==0)
                K[i][w] = 0;
            else if (wt[i-1] <= w)
                  K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
            else
                  K[i][w] = K[i-1][w];
        }
    }

    return K[n][W];
}
```

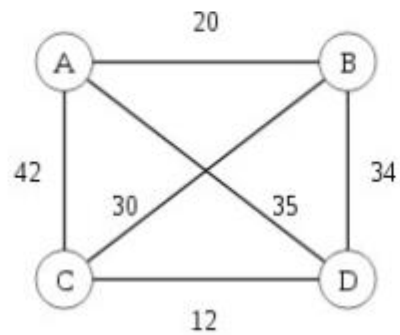
```c
int main(){
    int val[] = {100,70,50,10};
    int wt[] = {10,4,6,12};
    int  W = 12;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

	W=0	1	2	3	4	5	6	7	8	9	10	11	12
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	100	100	100
2	0	0	0	0	70	70	70	70	70	70	100	100	100
3	0	0	0	0	70	70	70	70	70	70	120	120	120
4	0	0	0	0	70	70	70	70	70	70	120	120	120

Traveling Salesman Problem(TSP)

- ปัญหานี้กำหนด n เมืองและระยะทางของทุกคู่ของเมืองในรูปของ matrix ที่ชื่อ dist มีขนาด nxn จงคำนวณค่าใช้จ่ายในการสร้าง tour ที่เริ่มที่เมือง s ใดๆ ผ่านทุกเมืองที่เหลือ n-1 เพียงครั้งเดียว แล้วสุดท้ายกลับมาที่เมืองเริ่มต้น s
- tour แบบนี้เรียกว่า Hamiltonian tour ซึ่งเป็น cycle ใน undirected graph ที่ผ่านแต่ละโหนดเพียงครั้งเดียวและกลับมาที่โหนดเริ่มต้น

- **ตัวอย่างกราฟต่อไปนี้ มี 4 เมือง**



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

ดังนั้นเราจะมี tour ได้ 4! = 24 แบบ(จาก permutation ของ 4)

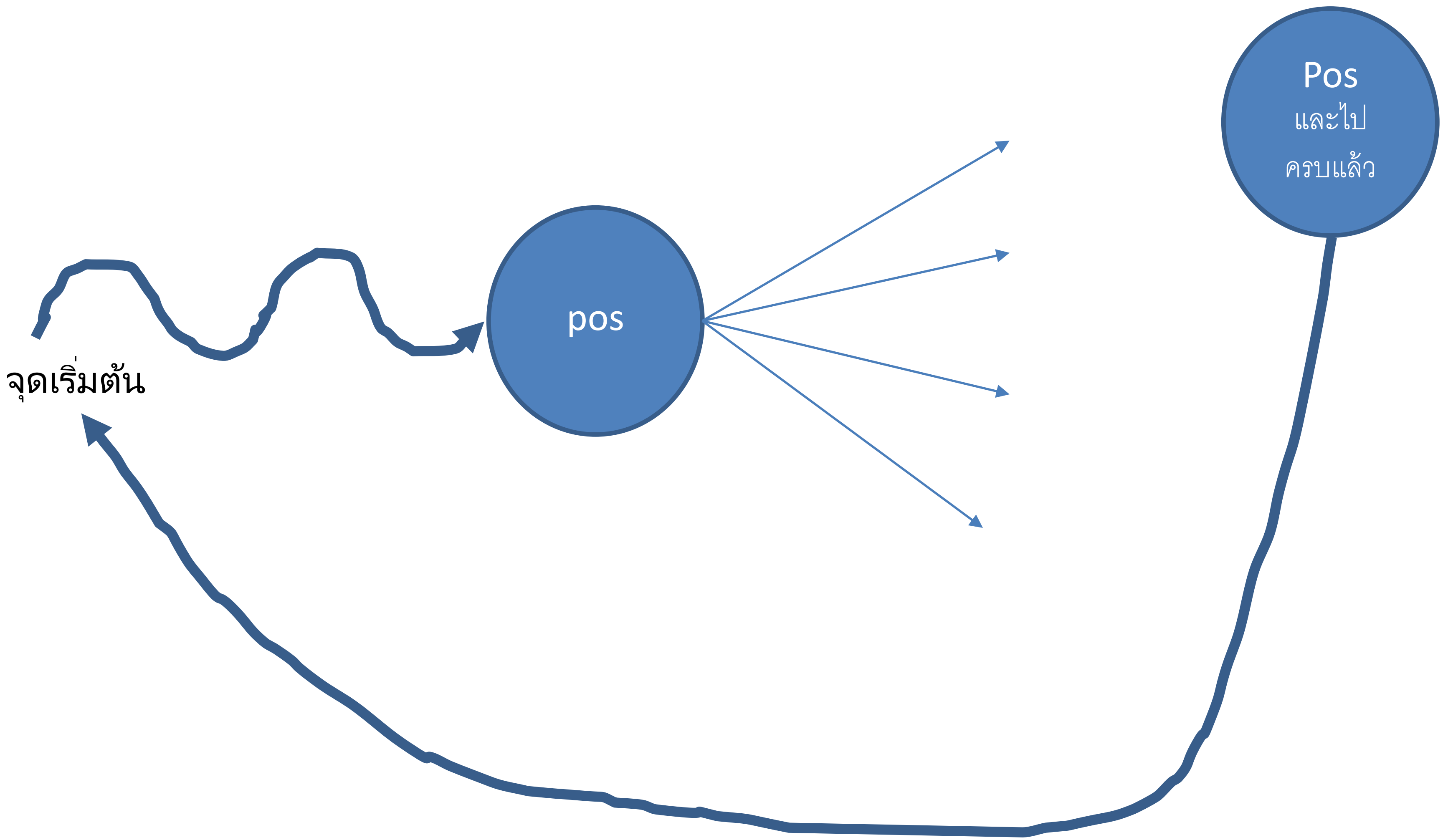
หนึ่งใน tour ที่ผลรวมต่ำสุดคือ A-B-C-D-A ซึ่งมีค่าใช้จ่าย

20+30+12+35 = 97

อาจจะมีคำตอบที่ดีที่สุดได้หลายแบบ

- วิธีการ brute force ของ TSP (ไม่ว่าจะ iterative หรือ recursive) ก็เป็นการพยายามลอง tour ทั้ง O((n−1)!) แบบ โดยการ fix โหนด A ไว้ก่อน
- วิธีนี้ใช้ได้ถ้า n มีค่าไม่เกิน 12 นั่นคือ 11! ประมาณ 40M เมื่อ n = 12 ก็จะนานเกิน
- อย่างไรก็ตาม ถ้ามีหลาย testcase การใช้ brute force ของ TSP ก็น่าจะเวลาเกิน

- เราสามารถใช้ DP กับ TSPได้เนื่องจาก การคำนวณ sub-tour นั้นซ้ำซ้อน เช่น tour A-B-C-(n-3) เมืองอื่นๆ ที่สุดท้ายแล้วกลับมาที่ A นั่นซ้ำกับ A-C-B-(n-3) เมืองเดิมที่สุดท้ายกลับมาที่ A
- ถ้าเราหลีกเลี่ยงการคำนวณความยาวของ subtour นั้นได้ เราก็จะประหยัดเวลาในการคำนวณนั่นเอง
- อย่างไรก็ตาม state ต่างกันใน TSP ขึ้นกับ 2 parameter : เมืองล่าสุดที่ผ่านมา pos และ subset ของโหนดที่แวะแล้ว



pos
Pos
และไป
ครบแล้ว
จุดเริ่มต้น

- มีหลายวิธีในการแทน set อย่างไรก็ตามเนื่องจากเราต้องการส่งผ่านข้อมูล set นี้เป็น parameter ของ recursive function (ถ้าใช้ top-down DP) เราต้องการการแสดงที่มีประสิทธิภาพและง่าย
- เราจะใช้ bitmask ถ้าเรามี n เมืองเราใช้ binary integer ยาว n โดยถ้า bit i เป็น 1 เราจะบอกว่าเมือง i นี้อยู่ใน set (นั่นคือยังถูก visit) และเป็น 0 ถ้าไม่อยู่ใน set
- ตัวอย่างเช่น mask = $18_{10} = 10010_2$ หมายความว่าเมือง {1,4} อยู่ใน set (ที่เรา visit แล้ว) เริ่มที่ 0 และนับจากทางขวา
- ในการเชค bit ที่ i ว่าเปิด(1) หรือปิด(0) เราใช้ mask & (1<<i)
- ในการเซต bit ที่ i เราสามารถใช้ mask |=(1<<i)

- แนวทางการแก้
- ใช้ complete search recurrence สำหรับ tsp(pos,mark):

1. tsp(pos,2^n–1) = dist[pos][0] นั่นคือทุกเมืองถูก visit แล้วกลับไปที่ เมืองเริ่มต้น mask = (1<<n)–1 หรือ 2^n–1 คือทุก n เมือง mask เป็น 1

2. tsp(pos,mark) = min(dist[pos][nxt]+tsp(nxt,mask|(1<<nxt))) สำหรับทุก nxt ที่อยู่ใน [0, n–1] และ nxt!= pos และ mask & (1<<nxt)) เป็น 0
   นั่นคือเราลองทุกเมืองต่อไป (nxt) ที่ยังไม่ได้ visit มาก่อน

- มีเพียง $O(nx2^n)$ state ที่แตกต่างกันเพราะว่ามี n เมืองและเราจำมากสุด 2^n เมืองที่เหลือที่ถูก visit ในแต่ละรอบ
- แต่ละ state สามารถถูกคำนวณได้ใน O(n) ดังนั้นเวลาทั้งหมดในการคำนวณ DP เป็น $O(2^n \times n^2)$ ทำให้เราแก้ปัญหาได้ n ประมาณ 16 นั่นคือ $16^2 x 2^{16}$ ประมาณ 17 M
- คำตอบหาได้จากการเรียก tsp(0,1) นั่นคือเริ่มที่เมือง 0 และ set mask = 1 นั่นจะทำให้เมือง 0 ไม่ถูกแวะอีก

- โดยทั่วไปแล้ว SP TSP ส่วนใหญ่ต้องการให้เรา preprocess graph ก่อนเพื่อทำ distance matrix ก่อนทำ DP

```
int tsp(int pos, int bitmask) {
// bitmask stores the visited coordinates
  if (bitmask == (1 << (n + 1)) - 1)
    return dist[pos][0]; // return trip to close the loop
  if (memo[pos][bitmask] != -1)
    return memo[pos][bitmask];

  int ans = 2000000000;
  for (int nxt = 0; nxt <= n; nxt++) // O(n) here
    if (nxt != pos && !(bitmask & (1 << nxt)))
// if coordinate nxt is not visited yet
      ans = min(ans, dist[pos][nxt] + tsp(nxt, bitmask | (1 << nxt)));
  return memo[pos][bitmask] = ans;
}
```