Dynamic Programming:
Principles and Applications

ภิญโญ แท้ประสาทสิทธิ์
ภาควิชาคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยศิลปากร
(pinyotae at gmail . com; pinyo at su.ac.th)

หัวข้อเนื้อหา

- อะไรคือไดนามิกโปรแกรมมิง (แบบผิวเผิน)
- ประโยชน์ของไดนามิกโปรแกรมมิง
- ตัวอย่างปัญหาที่แสดงแนวคิดและลักษณะเด่นของไดนามิกโปรแกรมมิง
- อะไรคือไดนามิกโปรแกรมมิง (แบบจริงจัง)
- จะรู้ได้ไงว่าจะใช้ไดนามิกโปรแกรมมิงได้หรือเปล่า
- กระบวนท่ามาตรฐาน
  - กระบวนท่าในยุคดั้งเดิม (classical techniques)
  - กระบวนท่าในยุคสมัยใหม่ (modern techniques)

อะไรคือไดนามิกโปรแกรมมิง (แบบผิวเผิน)

- เป็นวิธีการที่แก้ปัญหาด้วยการแบ่งปัญหาใหญ่ออกเป็นปัญหาย่อย
- ที่จริง Divide & Conquer ก็แบ่งปัญหาออกเป็นส่วนที่เล็กลง เพียงแต่ในไดนามิกโปรแกรมมิงนั้น ...
  - ปัญหาที่ใหญ่กว่าสองอันอาจจะมีปัญหาย่อยอันเดียวกัน (overlapping subproblem, ปัญหาย่อยคาบเกี่ยวกัน)
  - ถ้าปัญหาย่อยเป็นอันเดียวกัน เราแก้มันแค่ครั้งเดียวก็พอ แต่เอาคำตอบเดิมไปใช้กับปัญหาใหญ่ได้หลายรอบ → กำไรเห็น ๆ
  - มีโครงสร้างย่อยที่พึงปรารถนาที่สุด (optimal substructure)

ความคาบเกี่ยวกันกับ Greedy Algorithm

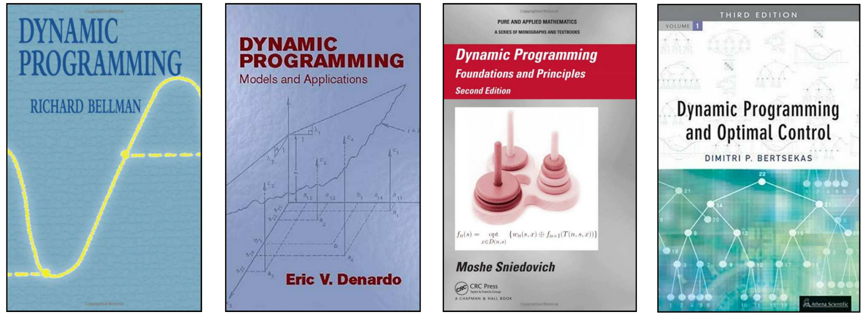
- ไดนามิกโปรแกรมมิงอาจจะถือได้ว่าเป็นสิ่งที่ซับซ้อนที่สุดเมื่อเทียบกับ Divide & Conquer และ Greedy Algorithm
  - มีเรื่องของ subproblem เช่นเดียวกับ Divide & Conquer
  - มีเรื่องของ optimal substructure เช่นเดียวกับ Greedy Algorithm
- บางปัญหาอาจถูกมองว่าเป็นทั้ง Greedy Algorithm และ Dynamic Programming เพราะกระบวนการคิดต้องทำผ่านทาง optimal substructure แต่ overlapping subproblem อาจจะไม่ใช่สิ่งที่เด่นชัด
  - Dijkstra’s Algorithm ก็มักถูกมองเป็นทั้งสองแบบ
  - แต่ที่แน่ ๆ ก็คือเราต้องมองเห็นว่าคำตอบที่ดีที่สุดของปัญหาหลักหาได้จากการนำคำตอบที่ดีที่สุดของปัญหาย่อยหลาย ๆ อันมาพิจารณาร่วมกัน

ประโยชน์ของไดนามิกโปรแกรมมิง

- เรามักใช้ไดนามิกโปรแกรมมิงกับปัญหาการคำนวณค่าสูงสุดหรือค่าต่ำสุด (ปัญหาพวก Optimization)
- คำว่าโปรแกรมมิงในที่นี้มาจาก Mathematical Programming แปลว่า Mathematical Optimization ไม่เกี่ยวกับการเขียนโปรแกรม
- มีการใช้กันอย่างแพร่หลายในหลายวงการ เช่น
  - การวางแผนการผลิตที่ทำให้ใช้เวลาน้อยที่สุด (Industrial Engineering)
  - การหาเส้นทางที่สั้นที่สุดสำหรับหุ่นยนต์ (Mechanical Engineering)
  - การทำนายเหตุการณ์ที่น่าจะเป็นที่สุด [ในบริบทของ Markov information sources] (Computer Science and Engineering, also several scientific and engineering fields that needs prediction)

ความสำคัญของไดนามิกโปรแกรมมิง

- ถือว่าสูงส่งมาก ถึงกับมีหนังสือจำนวนมากที่เขียนมาเพื่อเรื่องนี้โดยเฉพาะ



360 หน้า | 240 หน้า | 624 หน้า | 1022 หน้า

- นักเรียนสาขา Industrial Engineering ในระดับโท-เอกมักจะได้เรียนวิชาที่ชื่อว่า 'ไดนามิกโปรแกรมมิง' (อะไรมันจะเจาะจงขนาดนั้น)

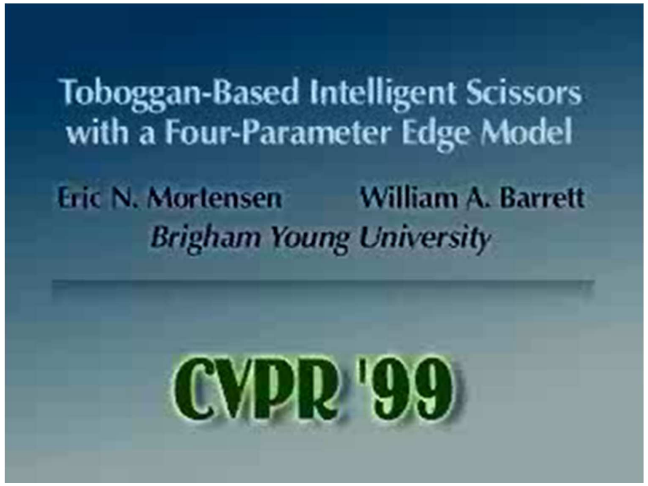
การประยุกต์ใช้ที่มีชื่อเสียง (แต่เราอาจไม่รู้ตัว)

- ที่จริงแล้ว Photoshop's Magnetic Lasso Tool มีการใช้ Dijkstra's Algorithm อยู่ภายใน
  - Magnetic Lasso มีพื้นฐานมาจาก Live wire ที่คิดค้นโดย Mortensen ในปี 1994 (สมัยนั้นถูกเรียกว่า 'Intelligent Scissors)
  - Live wire อาศัยหลักการพื้นฐานว่า (เส้นขอบวัตถุควรจะวิ่งผ่านบริเวณที่เป็นขอบให้มากที่สุด แต่ก็ไม่ควรจะวิ่งอ้อมมากเกินไป)
    - บริเวณที่เป็นขอบชัดเจนจะเป็นเหมือนทางที่สั้น
    - บริเวณที่ดูยากว่าเป็นขอบหรือไม่จะเป็นเหมือนทางที่ยาว
    - เราจะหาทางที่สั้นที่สุดระหว่างจุดที่ผู้ใช้กำหนดสองจุด โดยความสั้นยาวพิจารณาจากความชัดของขอบแทนที่จะเป็นจำนวนพิกเซล

ทัศนา Live Wire จากผู้คิดค้น

Image: Mortensen SIGGRAPH 1995

Live Wire in Action



http://web.engr.oregonstate.edu/~enm/publications/CVPR_99/toboggan_scissors.mov

หัวข้อเนื้อหา

- อะไรคือไดนามิกโปรแกรมมิง (แบบผิวเผิน)
- ประโยชน์ของไดนามิกโปรแกรมมิง
- **ตัวอย่างปัญหาที่แสดงแนวคิดและลักษณะเด่นของไดนามิกโปรแกรมมิง**
- อะไรคือไดนามิกโปรแกรมมิง (แบบจริงจัง)
- จะรู้ได้ไงว่าจะใช้ไดนามิกโปรแกรมมิงได้หรือเปล่า
- กระบวนท่ามาตรฐาน
  - กระบวนท่าในยุคดั้งเดิม (classical techniques)
  - กระบวนท่าในยุคสมัยใหม่ (modern techniques)
- การค้นคืนผลเฉลย

ตัวอย่างปัญหาที่ดูธรรมดาแต่ไม่ธรรมดา

ปัญหาเลือกซื้อชุดหรูไปงานแต่งงาน

- ต้องการซื้อชุด**ที่แพงที่สุด**ไปงานแต่ง ชุดมีหลายส่วน (C ส่วน) ตั้งแต่รองเท้า เข็มขัด เสื้อ กระโปรง สร้อย ฯลฯ
- แต่ละส่วนก็มีให้เลือกหลายแบบ เช่น รองเท้ามีให้เลือก 3 แบบ เสื้อมีให้เลือก 10 แบบ ต้องการเลือกซื้อทุกส่วนอย่างละ 1 แบบเพื่อมารวมเป็นชุดเดียว ภายใต้งบประมาณทั้งหมด (M)
- อินพุต: $1 \leq M \leq 200$ และ $1 \leq C \leq 20$ ต่อจากนั้นก็จะเป็นลิสต์ราคาของส่วนเสื้อผ้าทีละส่วน โดยแต่ละส่วนมีแบบให้เลือกอยู่ทั้งหมด $K_1, K_2, K_3, \ldots, K_C$ แบบ โดยที่ $1 \leq K_i \leq 20$
- ผลลัพธ์: **ราคา**ชุดที่แพงที่สุดหรือคำว่า no solution เมื่องบประมาณไม่พอ

ตัวอย่างอินพุตและผลลัพธ์

M = 20, C = 3

เสื้อผ้าส่วนที่หนึ่ง มี 3 แบบให้เลือกราคา 6 4 8

เสื้อผ้าส่วนที่สอง มี 2 แบบให้เลือกราคา 5 10

เสื้อผ้าส่วนที่สาม มี 4 แบบให้เลือกราคา 1 5 3 5

[ผลลัพธ์: ชุดที่แพงที่สุดที่ซื้อได้ราคา = 8 + 10 + 1 = 19]

M = 9, C = 3

เสื้อผ้าส่วนที่หนึ่ง มี 3 แบบให้เลือกราคา 6 4 8

เสื้อผ้าส่วนที่สอง มี 2 แบบให้เลือกราคา 5 10

เสื้อผ้าส่วนที่สาม มี 4 แบบให้เลือกราคา 1 5 3 5

[ผลลัพธ์: no solution งบไม่พอแม้แต่ชุดที่ถูกที่สุด]

มาดูวิธีแก้ปัญหาแบบต่าง ๆ กันก่อน

วิธีแรก Complete Search → Time Out / Time Limit Exceeded

- คำนวณทุกรูปแบบการจัดชุดที่เป็นไปได้
- แบบนี้อาการหนัก เพราะถ้ามี 20 ส่วนและแต่ละส่วนมี 20 แบบ จำนวนรูปแบบชุดที่เป็นไปได้คือ 20^{20}
- บางทีเราอาจจะคิดเพิ่มความเร็วขึ้นไป เป็นต้นว่าถ้าจัดไปได้สัก 5 ส่วน แล้วพบว่าเกินงบ ก็ไม่ต้องพยายามจัดต่อจากแบบที่เกินงบไปแล้ว
  - กรณีที่เลวร้ายที่สุดก็ยังต้องคำนวณทั้ง 20^{20} แบบอยู่ดี เพราะอาจจะไม่มีแบบที่เกินงบ
  - ต่อให้ลดการคำนวณลงไปได้สักล้านเท่า แต่วิธีนี้ก็มักจะใช้เวลานานเกินไปอยู่ดี เพราะ $\frac{20^{20}}{1000000} = 2^{20} \times 10^{14} \approx 10^{15}$ (พันล้านล้านรูปแบบ)

ฝึกซ้อมเขียนแบบ Complete Search ให้ดี

ถึงเราจะรู้ว่าวิธีแบบ Complete Search/Backtracking มันช้า

- แต่มันก็ช่วยให้เราใช้ตรวจคำตอบวิธีที่เร็ว ๆ ได้
- ช่วยพัฒนาทักษะการเขียนโปรแกรมอย่างคาดไม่ถึง (เดี๋ยวได้เห็นกันว่ามันไม่ง่าย)
- ทักษะที่ว่านำไปสู่กระบวนการแก้ปัญหาแบบไดนามิกโปรแกรมมิงที่สวยมาก

คำถาม เราจะเขียนโปรแกรมแบบจับรูปแบบทุกอันที่เป็นไปได้ด้วยวิธีไหน ?

- อย่าลืมว่าจำนวนส่วนของชุดมันไม่แน่นอน เรารู้แค่ว่ามีอยู่ 1 – 20 แบบ
- จะเขียนลูป 20 ชั้นเตรียมไว้เลยมั้ย ถ้า C มันน้อยเราก็ไม่ต้องใช้ลูปด้านใน
- เอ เขียนแบบนั้นมันดูประหลาด แต่ถ้าไม่ทำลูปเตรียมไว้ จะทำไงดี ?

Complete Search แบบ Recursive

- ในกรณีที่เราต้องทำการวนลูปแบบเดิม ๆ แค่เปลี่ยนข้อมูล แต่ก็ไม่รู้ว่าจะรันกี่ชั้นดี → อาการแบบนี้มันเรียกร้องให้เราใช้วิธีแบบ Recursive มาก
  - ถ้าไม่ใช้จะยากกว่าเดิม (ต้องคอยสลับค่าตัวแปรควบคุมลูปไปมาเอง)
- แต่ปัญหาคือพารามิเตอร์ของฟังก์ชัน recursive ที่เหมาะสมคืออะไร ?
  - เนื่องจากเราจะใช้ recursive ทดแทนลูปแต่ละชั้น ดังนั้นตัวแปรที่จะใช้ก็ควรจะเกี่ยวกับการระบุว่าเราจะจัดการลูปชั้นไหน (แต่ละชั้นแทนส่วนของชุด)
  - เนื่องจากเราต้องการตัดกรณีที่เป็นไปไม่ได้บางอันออก (เกินงบ) เราจะใช้ค่าใช้จ่ายรวมตามแบบของชุดที่เลือกมาก่อนหน้าเป็นตัวช่วย
  - การตัดคำตอบแบบนี้ออกเรียกว่า backtracking
  - สองพารามิเตอร์นี้นำไปสู่ฟังก์ชันในการแก้ปัญหา solve(int part, int curCost);

ฟังก์ชัน solve สำหรับ Complete Search (Recursive)

```
int bestCost;
void solve(int part, int curCost) {
    if(part == C) {
        if(curCost > bestCost)
            bestCost = curCost;
        return;
    }

    for(int i = 0; i < arK[part]; ++i) {
        int newCost = curCost + arCost[part][i];
        if(newCost > M) {
            return;
        } else {
            solve(part + 1, newCost);
        }
    }
}
```

ฟังก์ชัน main สำหรับ Complete Search

```
int main() {
    scanf("%d %d", &M, &C);
    for(int i = 0; i < C; ++i) {
        scanf("%d", &arK[i]);
        for(int j = 0; j < arK[i]; ++j) {
            scanf("%d", &arCost[i][j]);
        }
    }
    bestCost = -1;
    solve(0, 0);
    if(bestCost == -1) printf("no solution");
    else printf("%d", bestCost);

    return 0;
}
```


ลองใช้ Greedy Algorithm แก้ปัญหา

หากเราจะเลือกส่วนที่แพงที่สุดเท่าที่งบเหลืออยู่ มันก็จะ**<u>นำไปสู่คำตอบที่ผิด</u>**

M = 12, C = 3

เสื้อผ้าส่วนที่หนึ่ง มี 3 แบบให้เลือกราคา 6 4 8

เสื้อผ้าส่วนที่สอง มี 2 แบบให้เลือกราคา 5 10

เสื้อผ้าส่วนที่สาม มี 4 แบบให้เลือกราคา 1 5 3 5

ถ้าใช้ Greedy ก็จะเลือกแบบที่ราคา 8 หน่วยมาตอนแรก และก็จะพบว่างบประมาณที่เหลือไม่พอที่จะซื้อส่วนที่สองและก็จะตอบมาว่า no solution

<u>แต่คำตอบแท้จริงก็คือว่าเราได้ชุดที่ราคา 6 + 5 + 1 = 12 หน่วย</u>

วิธีที่สาม: ไดนามิกโปรแกรมมิง

- เราจะพยายามแก้ปัญหาด้วยการหาว่า**ด้วยเงินที่ใช้ไปและของที่เลือกไปแล้ว เราจะเลือกซื้ออะไรต่อไปถึงจะดีที่สุด**
  - สังเกตได้ว่า ไม่ว่าจะได้ชุดที่สมบูรณ์แบบไหนเราต้องพิจารณาทุกส่วน นั่นคือเราต้องมีการพิจารณาชุดแต่ละส่วนซ้ำ ๆ กัน
  - สิ่งที่สังเกตออกยากก็คือว่า มันมีโอกาสที่เลขงบที่เหลืออยู่จะซ้ำกันได้ด้วย
  - เช่น M = 200 และเราซื้อส่วนที่หนึ่งและสองด้วยเงิน 50 และ 30 หน่วย กับซื้อส่วนที่หนึ่งและสองด้วยเงิน 20 และ 60 หน่วย
    - แบบนี้แสดงว่าไม่ว่าเลือกสองส่วนแรกมาด้วยวิธีไหนที่ใช้เงินเท่ากัน ขั้นตอนต่อมาก็จะคำนวณด้วยวิธีเดียวกันได้ (เพราะเหลืองบเท่ากัน)
    - แบบนี้แหละที่เราเรียกว่าปัญหาย่อยมันคาบเกี่ยวกันและเราใช้คำตอบเดิมในปัญหาย่อย ไปตอบปัญหาที่ใหญ่กว่าได้หลายครั้งโดยไม่ต้องคำนวณใหม่

ขอแบบชัด ๆ ว่าปัญหาย่อยคืออะไร

- จากตัวอย่างที่ยกมานั้น เราต้องการทราบว่า **หลังจากซื้อชุดสองส่วนแรกไปแล้วเป็นเงิน 80 หน่วยเราจะซื้อส่วนที่เหลืออย่างไรถึงจะใช้เงินได้เต็มงบที่สุด**
  - ตอนแรกเราเริ่มด้วยการถามว่า เมื่อยังไม่ซื้อชุดไปสักส่วน (0 ส่วนแรก) เราใช้เงินไปเป็น 0 หน่วย เราจะซื้อส่วนที่เหลืออย่างไรถึงจะใช้เงินได้เต็มงบที่สุด
  - พอจะเห็นภาพแล้วใช่หรือไม่ว่าจริงแล้ว**เราถามคำถามคล้าย ๆ เดิมจากปัญหาเริ่มต้น** เพียงแต่วงเงินและจำนวนส่วนของชุดมันลดลง
  - แสดงว่าเมื่อทำการย่อยปัญหาแล้ว ปัญหาย่อยจะมีค่า M และ C ลดลง

- คำถามที่ตามก็คือ มันคาบเกี่ยวกันยังไง และสุดท้ายก็คือทำแบบนี้แล้วเราจะได้คำตอบที่ดีที่สุดจริงหรือ

ดูอีกทีว่าปัญหาย่อยคาบเกี่ยวกันอย่างไร

- ปัญหาย่อยคาบเกี่ยวกันถ้าคำตอบของปัญหาย่อยอันหนึ่งใช้ได้มากกว่าหนึ่งครั้งกับปัญหาใหญ่ที่ไม่เหมือนกัน
  - ปัญหาที่ไม่เหมือนกันหมายความว่า ปัญหามันอาศัยเส้นทางในการตามหาคำตอบคนละทาง
    - เช่น จะหาชุดที่ได้จากการซื้อสองส่วนแรกด้วยเงิน 50 และ 30 หน่วยก็เป็นทางหนึ่ง ชุดที่ได้จากการซื้อสองส่วนแรกด้วยเงิน 20 และ 60 หน่วยก็เป็นอีกทางหนึ่ง
  - แต่เราจะเห็นได้ว่า ในเมื่อมันใช้เงินไปเท่ากัน งบที่เหลือก็เท่ากัน การคำนวณในส่วนที่เหลือก็จะเหมือนกันทุกอย่าง
    - แสดงว่าถ้าหาคำตอบของการซื้อชุดส่วนที่เหลือด้วยงบอีก 120 หน่วยได้เราก็จะตอบคำถามของปัญหาทั้งสองได้พร้อมกัน

แล้วคำตอบจะถูกหรือเปล่า

- อันนี้เป็นประเด็นที่สำคัญมาก เพราะหากคำตอบที่ได้จากปัญหาย่อย ไม่ได้ช่วยให้เราได้คำตอบที่ถูกต้อง ก็แสดงว่าวิธีเรามันผิดไปจากจุดประสงค์
- วิธีพิจารณาว่าคำตอบจะถูกต้องหรือเปล่ามีหลักการทางคณิตศาสตร์บางอย่างกำกับไว้อยู่ แต่เราจะพูดถึงมันแบบไม่เป็นทางการก่อน
  - ถ้าหากปัญหาย่อยที่แก้ได้ สามารถนำมาใช้ประกอบเป็นคำตอบของปัญหาที่ใหญ่ขึ้นได้ ผ่านกระบวนการพิจารณาเพิ่มเติมบางอย่าง แสดงว่าปัญหามี optimal substructure
  - กระบวนการพิจารณาเพิ่มเติมที่ว่ามักเป็นการคัดสรรคำตอบจากปัญหาย่อยบวกกับอินพุตเพิ่มเติมในปัญหาใหญ่เพื่อเลือกชุดคำตอบที่ดีที่สุด

การคัดสรรผลเฉลยย่อย

- ส่วนใหญ่การคัดสรรผลเฉลยย่อยจะกระทำโดยการนำผลเฉลยย่อยในระดับที่ต่ำลงมาหนึ่งระดับมาลองบวกด้วยค่าอินพุตในระดับที่กำลังพิจารณาอยู่
  - จากนั้นก็คัดเอาผลบวกที่อาจจะดีที่สุดไปใช้เป็นผลเฉลยย่อยในระดับที่สูงขึ้น
  - ทำแบบนี้ซ้ำไปเรื่อย ๆ จนได้ผลเฉลยย่อยที่ระดับสูงสุดซึ่งก็คือผลเฉลยปัญหา
- สำหรับตัวอย่างนี้เราจะนำผลรวมยอดเงินที่ใช้จากการซื้อส่วนที่สามของชุดแต่ละอันที่ไม่เกินงบมาบวกกับราคาชุดส่วนที่สอง
  - หากผลการบวกไม่เกินงบเราก็เก็บเป็นผลลัพธ์ยอดเงินหลังจากซื้อส่วนที่สองและสามไว้ ถ้าหากเกินงบเราก็ไม่ต้องเก็บ
  - การเลือกเก็บผลลัพธ์ที่ถูกต้องไว้แบบนี้ก็เพราะมันยังเร็วเกินไปที่จะบอกได้ว่าใครจะนำไปสู่คำตอบที่ดีที่สุด เราจึงเก็บคำตอบที่ถูกต้องเอาไว้ใช้เท่านั้น

เกิดอะไรขึ้นบ้างในการคำนวณ

M = 12, C = 3

เสื้อผ้าส่วนที่หนึ่ง มี 3 แบบให้เลือกราคา 6 4 8

เสื้อผ้าส่วนที่สอง มี 2 แบบให้เลือกราคา 5 10

เสื้อผ้าส่วนที่สาม มี 4 แบบให้เลือกราคา 1 5 3 5

- เมื่อเลือกซื้อส่วนที่สาม ยอดเงินที่ใช้ไปก็คือ 1, 3, 5
- เมื่อเลือกซื้อส่วนที่สองเพิ่มเติม ยอดเงินที่ใช้ไปคือ 6, 8, 10, 11
- ยอดเกินงบไปแล้วถูกตัดออก ซึ่งก็คือยอดเงินรวมที่ 13 และ 15
- เมื่อเลือกซื้อส่วนแรกเพิ่มเติม ยอดเงินที่ใช้ไปคือ 10 และ 12

ดังนั้นสรุปได้ว่าชุดที่แพงที่สุดที่ซื้อได้ราคา 12 หน่วย

- สังเกตให้ดีว่าในระหว่างการคำนวณมันจะมียอดเงินซ้ำเกิดขึ้นบ่อย ๆ แต่เราไม่ต้องคิดซ้ำ

แล้วตกลงทำไมมันถูกและมันเร็วกว่าวิธีเดิมจริงหรือ

- ถูกต้องก็เพราะวิธีนี้เราเก็บยอดเงินทั้งหมดที่เป็นไปได้ไว้ โดยไม่ทิ้งออกไปเลย → ไม่เปิดโอกาสให้ผิด
  - เราไม่ปล่อยให้ผลเฉลยย่อยที่อาจจะนำไปสู่คำตอบที่ดีที่สุดหลุดลอยไป
  - เมื่อเลือกซื้อชุดทุกส่วนไปแล้วตัวเลขที่ได้แท้จริงก็คือยอดเงินทั้งหมดที่เป็นไปได้ในการซื้อทั้งชุดภายใต้งบประมาณที่กำหนด
  - เราเลือกตัวที่ดีที่สุดมาก็จะได้คำตอบสุดท้ายทันที
- มันเร็วกว่าเพราะในการเลือกซื้อเพิ่มแต่ละส่วน …
  - จำนวนรูปแบบมีอย่างมาก M เราดูแค่ระดับล่าสุดก็พอ ไม่ต้องดูย้อนไปไกล
  - ดังนั้นเมื่อรวมทุกชั้นอย่างมากเราก็คือ M x C x K
  - กรณีที่แย่ที่สุดคือ 200 x 20 x 20 = 80,000 → น้อยกว่าวิธีเดิมมาก

แล้วจะเขียนโปรแกรมยังไงดี

- ปัญหาแรกที่เจอก็คือ ‘จะรู้ได้ไงว่าปัญหาย่อยอันไหนที่แก้ไปแล้ว’
  - แก้ด้วยการสร้างตารางเก็บคำตอบไว้ ถ้ามีคำตอบอยู่แล้วก็หยิบใช้เลย

	0	1	2	3	4	5	6	7	8	9	10	11	12
1	T												
2			T		T	T	T		T		T	T	
3				T		T							

  - ถ้ายังไม่มีคำตอบของปัญหาย่อยก็ให้คำนวณคำตอบขึ้นมา
  - แต่เราคำนวณคำตอบของปัญหาย่อยได้ก็ต่อเมื่อมันเป็นระดับล่างสุด หรือคำตอบของปัญหาย่อยระดับก่อนหน้ามีคำตอบเตรียมไว้ให้ใช้อยู่แล้ว

การเตรียมคำตอบของปัญหาย่อย

- มีอยู่สองแนวทางคือคือ
  - แบบเตรียมเฉพาะที่จำเป็นจริง ๆ (แบบ Top-Down)
  - แบบเตรียมล่วงหน้าไว้ให้ครบโดยสมบูรณ์ (แบบ Bottom-Up)
- แบบ Top-Down จะมีธรรมชาติเป็น recursive หลายคนจะไม่ถนัดคิดแนวนี้ แต่โค้ดมักจะสั้นกว่ามีโอกาสเขียนผิดน้อยกว่า (ถ้าเราคิดวิธีออก)
  - ถ้าใครเขียนพวก complete search/backtracking สำเร็จ มักจะเขียนแบบ top-down เสร็จเร็วมาก
- แบบ Bottom-Up จะมีธรรมชาติเป็น iterative มักจะเป็นลูปซ้อนกันแค่สองหรือสามชั้นตามรูปแบบตารางที่ใช้เก็บคำตอบของปัญหาย่อย
  - คนส่วนใหญ่มาแนวนี้ แต่โค้ดมักจะยาวขึ้นหรือมีโอกาสเขียนผิดมากกว่า

แนวคิดแบบ Top-Down

ถามหาว่าเริ่มจากส่วนแรกของชุดโดยยังไม่ได้ใช้เงินไปเลย แล้วคำตอบที่ดีที่สุดคืออะไร ในที่นี้สมมติว่าเราหาคำตอบด้วยฟังก์ชัน solve(part, cost)

- เราพยายามหาคำตอบของ solve(0, 0); คือเริ่มจากส่วนแรกและยังไม่ได้ใช้เงิน แน่นอนว่า ณ เวลานี้เรายังหาคำตอบอะไรไม่ได้
- เราจึงนำอินพุตจากส่วนแรกของชุดซึ่งก็คือ ราคาของที่เลือกไว้มาบวกกับคำตอบจากปัญหาย่อย แล้วคัดผลลัพธ์ที่ดีที่สุด
- ประเด็นมันอยู่ที่ว่า เมื่อเลือกของแต่ละแบบไปแล้ว ยอดการใช้เงินก็เพิ่ม และเราก็ได้ส่วนของชุดเพิ่มขึ้นมาอีกหนึ่ง สมมติว่าของที่เลือกไปตอนแรกราคา 6 หน่วย แสดงว่า ณ ตอนนี้เราต้องการหาค่า solve(1, 6);
- ถ้าของชิ้นที่สองที่เราเลือกราคา 3 หน่วย เราก็ต้องการหาค่า solve(2, 9);

เขียนโค้ดแบบ Top-Down (1)

- สังเกตวิธีคิด จะได้ว่าเมื่อไปถึงเสื้อผ้าส่วนสุดท้ายเราก็จะได้คำตอบย่อยที่ส่งผลย้อนกลับขึ้นไปเป็นคำตอบใหญ่คำตอบเดียวได้
- หรือถ้าคำตอบมีอยู่แล้วเราก็ไม่ต้องคำนวณใหม่
- แต่จุดที่ได้มาซึ่งคำตอบย่อยอีกจุดหนึ่งก็คือ 'จุดที่รู้ว่าราคาเกินงบ'

ดังนั้นเราฟังก์ชัน solve จึงมี base case สามแบบดังแสดงข้างล่าง

```
int solve(int part, int curCost) {
    if(part == C)
        return curCost;
    if(curCost > M)
        return -1;
    if(memo[part][curCost] != -2)
        return memo[part][curCost];
    ……
}
```

ที่เก็บข้อมูลสำหรับปัญหาเลือกซื้อเสื้อผ้า

```
int M, C;          // พารามิเตอร์หลักของปัญหา
int arK[20];       // จำนวนแบบของเสื้อผ้าแต่ละส่วน
int arCost[20][20];    // ราคาของเสื้อผ้าแต่ละส่วนทุกแบบ
int memo[20][201];     // ที่เก็บผลลัพธ์
```

- ของพวกนี้ประกาศไว้เป็นตัวแปลแบบโกลบอลหรือ class member
  - ใน C และ C++ เราไม่ควรประกาศอาเรย์ใหญ่ ๆ ไว้ในฟังก์ชัน
- ประกาศอาเรย์ด้วยขนาดสูงสุดที่ปัญหากำหนดไว้
  - ทำให้เขียนโปรแกรมง่าย
  - แต่ถ้าปัญหาระบุขนาดหน่วยความจำมาให้เราแค่นิดเดียวก็ต้องระวัง อย่างไรก็ตามปัญหาที่มีเป้าหมายที่หน่วยความจำแบบนี้ถือว่ามีค่อนข้างน้อย

memo เอาไว้เก็บอะไรดี

การกำหนดความหมายของค่าใน memo ถือว่าสำคัญมาก

- ในที่นี้เราเลือกเก็บจำนวนเงินทั้งหมดที่ใช้ไปจากการซื้อเสื้อผ้าส่วนที่หนึ่งถึงส่วนที่ k <u>เมื่อค่าใช้จ่ายเริ่มต้นของแต่ละส่วนคือ m</u>
- เลข -2 เป็นค่าเริ่มต้น ระบุว่ายังไม่มีการเติมข้อมูลลงในตาราง
- เลข -1 แปลว่าไม่สามารถหาคำตอบได้ (ราคาเกินงบประมาณ)
- เลขบวกหมายความว่ามีคำตอบอย่างใดอย่างหนึ่งที่หาได้

	0	1	2	3	4	5	6	7	8	9	10	11	12
1	12	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
2	-2	-2	-2	-2	12	-2	12	-2	-2	-2	-2	-2	-2
3	-2	-2	-2	-2	-2	-2	-2	-2	-2	12	-2	12	-2

M = 12, C = 3

6 4 8

5 10

1 5 3 5

ใจความของฟังก์ชันคำนวณค่าชุด

วนหาค่าจากเสื้อผ้าทุกแบบในส่วนที่สนใจ

```
int bestCost = -1;

for(int i = 0; i < arK[part]; ++i) {
  int newCost = curCost + arCost[part][i];
  if(newCost <= M) {
    int returnCost = solve(part+1, newCost);
    if(returnCost > 0 && returnCost > bestCost) {
      bestCost = returnCost;
      memo[part][curCost] = returnCost;
    } else if(returnCost == -1 &&
         memo[part][curCost] == -2) {
      memo[part][curCost] = -1;
    }
  }
}

return bestCost;
```

สังเกตให้ดีว่าเราต้องเปรียบเทียบผลลัพธ์ที่ได้จากการเลือกซื้อแต่ละแบบและเก็บคำตอบที่ดีที่สุดไว้ใน memo แต่ละช่อง

เรื่องเล็กน้อยที่เราต้องแม่น

โครงสร้างข้อมูลและการกำหนดค่าเริ่มต้นต้องสอดคล้องกับวิธีการ

```
scanf("%d %d", &M, &C);
for(int i = 0; i < C; ++i) {
  scanf("%d", &arK[i]);
  for(int j = 0; j < arK[i]; ++j) {
    scanf("%d", &arCost[i][j]);
  }
}

for(int r = 0; r < 20; ++r) {
  for(int c = 0; c < 201; ++c) {
    memo[r][c] = -2;
  }
}
```

ค่าเริ่มต้นในอาเรย์ของ DP สำคัญมาก
ต้องมีความหมายสอดคล้องกับการใช้งาน

แนวคิดแบบ Bottom-Up

- หากเราจำการใช้ตารางของแบบ Top-Down ได้ เราจะพบว่าเรากำหนดค่าเริ่มต้นเป็น -2 เพื่อระบุว่ายังไม่มีการคำนวณคำตอบ
- แต่แนวคิดแบบ Bottom-Up จะเป็นไปในลักษณะที่ว่าเตรียมคำตอบสำหรับทุกกรณีที่อยู่ในตารางระดับล่างสุดเอาไว้ก่อนเลย
  - ไม่สำคัญว่าจะได้เรียกใช้จริงหรือไม่
  - รับประกันว่าตอนหาคำตอบในระดับที่สูงขึ้น มีคำตอบย่อยรอไว้ให้ใช้แน่นอน
  - เตรียมคำตอบทุกกรณีในระดับสูงขึ้นมา โดยสานต่อจากระดับล่างที่เตรียมไว้ → รับประกันเพิ่มขึ้นอีกระดับว่ามีคำตอบย่อยรอไว้ให้ใช้ในระดับสูงขึ้น
  - ทำไปเรื่อย ๆ จนถึงระดับสูงสุดก็จะได้คำตอบสุดท้าย

คำตอบของปัญหาย่อยในวิธีคิดแบบ Bottom-Up

- เป็นไปได้ที่ตาราง memo จะมีความหมายต่างจากแบบ Top-Down
- ในแบบ Bottom-Up นี้เราอาจจะเลือกแทนช่องแต่ละช่องเพื่อบันทึกว่ามีราคารวมของชุดส่วนที่หนึ่งถึงส่วนที่ k เท่าใดบ้าง
  - ถ้าหากมีราคารวมของชุดจากส่วนที่หนึ่งถึงส่วนที่ k ราคา m ช่อง memo[k][m] ก็จะถูกเซ็ตค่าให้เท่ากับ 1 ถ้าไม่อย่างนั้นก็มีค่า -1
  - เริ่มมาทุกช่องใน memo มีค่าเท่ากับ -1

	0	1	2	3	4	5	6	7	8	9	10	11	12
1	-1	-1	-1	-1	1	-1	1	-1	1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1	1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1	1

M = 12, C = 3
6 4 8
5 10
1 5 3 5

วิธีเติมค่าในตาราง

แถวแรกเราเติมเข้าไปด้วยราคาจากส่วนที่หนึ่งโดยตรง

	0	1	2	3	4	5	6	7	8	9	10	11	12
1	-1	-1	-1	-1	1	-1	1	-1	1	-1	-1	-1	-1

M = 12, C = 3
6 4 8
5 10
1 5 3 5

ส่วนแถวที่สองเราจะเลือกเฉพาะ m1 ในแถวแรกที่มีค่าเป็น 1 จากนั้นเราก็จะทำการบวกราคาชุดส่วนที่สองแต่ละอันเข้าไป

ได้ราคาเป็น m2 ถ้า $m2 \leq M$ ก็กำหนดค่าในช่อง m2 แถวที่สองเป็น 1

	0	1	2	3	4	5	6	7	8	9	10	11	12
1	-1	-1	-1	-1	1	-1	1	-1	1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1	1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1	-1	1

M = 12, C = 3
6 4 8
5 10
1 5 3 5

โค้ดเติมตารางแบบ Bottom-Up

```
for(int k = 0; k < (int) vP[1].size(); ++k) {
  int spend = vP[1][k];
  if(spend <= M)
    memo[1][spend] = 1;
}

for(int c = 2; c <= C; ++c) {
  for(int m = 0; m <= M; ++m) {
    if(memo[c-1][m] < 0)
      continue;
    for(int k = 0; k < (int) vP[c].size(); ++k) {
      int spend = m + vP[c][k];
      if(spend <= M)
        memo[c][spend] = 1;
    }
  }
}
```


ส่วนสรุปคำตอบแบบ Bottom-Up

เราวนหาค่าที่มากที่สุดในแถวสุดท้าย (k = C)

ถ้ามีชุดที่ซื้อได้ในงบประมาณจริงก็จะมีช่องใน memo ที่มีค่าเท่ากับ 1

```
int best = -1;
for(int m = 0; m <= M; ++m) {
    if(memo[C][m] > 0) best = m;
}
if(best <= M && best >= 0)
    printf("%d\n", best);
else
    printf("no solution\n");
```


แล้วชุดที่ดีที่สุดคืออะไร

- วิธีที่พูดมาทั้งหมดบอกแค่ว่าชุดที่แพงที่สุดเท่าไหร่
  - ยังไม่ได้หาว่าชุดที่ดีที่สุดมีอะไรเป็นองค์ประกอบบ้าง
- การหาชุดที่ดีที่สุดโดยปรกติจะทำควบคู่กันไปกับการหาค่าที่ดีที่สุด
  - มักจะต้องสร้างตารางอีกอันหนึ่งมาใช้ในการเก็บรูปแบบชุด
  - เราต้องคอยติดตามปรับค่าในตารางรูปแบบชุดในระหว่างหาค่าที่ดีที่สุด
- รูปแบบชุดที่ดีที่สุดอาจมีได้มากกว่าหนึ่งแบบ
  - ถ้าต้องการแบบใดแบบหนึ่งก็ได้ → ง่าย
  - ถ้าต้องการทุกชุดหรือชุดที่มีคุณสมบัติอย่างใดอย่างหนึ่งมันจะยากกว่าเดิม
- แต่ขั้นตอนในการหารูปแบบชุดเป็นแค่เรื่องของการเขียนโปรแกรมทั่วไป
  - ถ้ามาถึงจุดที่หาค่าที่ดีที่สุดได้จะทำตรงนี้ได้เอง (แต่อาจเสียเวลาไปบ้าง)

การค้นคืนรูปแบบชุด

ปัญหาที่ต้องการทดสอบความสามารถในการประยุกต์ใช้วิธีการ มักจะให้เราตอบออกมาในรูปที่นอกเหนือจากมูลค่าที่ดีที่สุด

- เช่น ต้องการให้เราระบุราคาของชิ้นส่วนแต่ละชิ้นในชุดที่หรูที่สุด
- แต่คำตอบก็มีหลากหลาย อาจจะตรวจด้วยเครื่องยาก เช่น ปัญหา ‘ลำดับเพิ่มขึ้นที่ยาวที่สุด’
- เพื่อให้ตรวจง่าย (และปัญหาสมจริงขึ้น) ตัวโจทย์มักจะถูกปรับเปลี่ยนเพื่อให้คำตอบเหลือแค่แบบเดียว
- อาจจะเป็นที่คาดไม่ถึงสำหรับคนเขียนโปรแกรมว่า ‘เปลี่ยนแค่นี้แต่เราต้องออกแรงคิดและเขียนโค้ดเพิ่มขึ้นมหาศาล’

การค้นคืนรูปแบบชุด (เวอร์ชันง่าย)

ตัวอย่าง: จงระบุราคาชิ้นส่วนแต่ละชิ้นในชุดที่หรูที่สุดเรียงตามลำดับประเภทจากลำดับแรกไปลำดับสุดท้าย ในกรณีที่ชุดหรูที่สุดมีหลายแบบ ให้เลือกตอบแบบใดแบบหนึ่ง

วิธีคิด: สร้างตารางอีกอันมาเก็บคำตอบไว้โดยเก็บไว้ว่าคำตอบในแต่ละแถวที่ r คอลัมน์ที่ c เป็นคำตอบที่สืบเนื่องจากคอลัมน์ที่ x ในแถวก่อนหน้า (r – 1)

เราคอยบันทึกค่า x นี้ในแต่ละช่อง ก็จะสืบย้อนกลับไปได้

(โครงสร้างข้อมูลที่ใช้เก็บคำตอบไม่ต้องเป็นตารางก็ได้ บางคนก็เลือกใช้ pair มาช่วยบันทึกคำตอบ แต่เชื่อได้ว่าส่วนใหญ่จะใช้ตาราง)

การค้นคืนรูปแบบชุด (เวอร์ชันใช้สมองขึ้นมาหน่อย)

ตัวอย่าง: จงระบุราคาชิ้นส่วนแต่ละชิ้นในชุดที่หรูที่สุดเรียงตามลำดับประเภทจากลำดับแรกไปลำดับสุดท้าย <u>ในกรณีที่ชุดหรูที่สุดมีหลายแบบ ให้ตอบออกมาเป็นจำนวนรูปแบบทั้งหมดของชุดหรูที่สุด</u>

หมายเหตุ เวลาในการคำนวณแทบไม่เปลี่ยน และรับรองได้ว่ามีคำตอบเดียว

วิธีคิด: ...

การค้นคืนรูปแบบชุด (เวอร์ชันใช้สมองขึ้นกว่าเดิมมาก)

ตัวอย่าง จงระบุราคาชิ้นส่วนของชุดที่หรูที่สุดแต่ละแบบ โดยเรียงราคาจากน้อยไปมาก ในกรณีที่ชุดหรูที่สุดมีหลายแบบให้เลือกแบบที่ชิ้นส่วนที่ราคาน้อยที่สุดมีราคาสูงที่สุด ถ้าชิ้นส่วนที่ราคาน้อยที่สุดมีราคาเท่ากันให้พิจารณาชิ้นส่วนที่มีราคาน้อยที่สุดถัดมา เป็นลักษณะนี้ไปเรื่อย ๆ

หมายเหตุ เวลาในการคำนวณอาจเพิ่มขึ้นมาก แต่รับรองได้ว่ามีคำตอบเดียว

วิธีคิด: ...

ถ้าขอบเขตของเงินกว้างมาก (M มีค่าเยอะ) จะทำไงดี

- ถ้า M มีค่าเยอะ ตารางเมโมก็จะใหญ่ขึ้นไปด้วย
- นั่นคือเวลาที่ต้องใช้ในการคำนวณก็มากขึ้นไปตาม อย่างน้อยที่สุดตอนจะ allocate อาเรย์ก็ใช้เวลาไปมากพอควร
- สมมติว่าความหลากหลายของราคาที่เป็นไปได้มีไม่มากนัก เช่น ประมาณ 10,000 ต่อระดับ ในขณะที่พิสัยของราคากว้างถึง 100 ล้าน
  - ถ้าใช้ bottom-up ถึงแม้ว่าเราจะไม่ต้องสร้างอาเรย์แบบเต็ม ๆ ขึ้นมาเพราะสร้างแค่สองระดับก็พอ แต่จะวิ่งทั้ง 100 ล้านก็มากเกิน
  - ถ้าใช้แบบ top-down ตาราง memo ก็จะใหญ่มาก เนื่องจากเราต้องวิ่งขึ้นลงหลายระดับ จะไม่เก็บไว้หลายระดับแต่แรกก็คงจะไม่ได้
  - ปัญหานี้มีทางออกแน่นอน แต่เราควรทำอย่างไร

โครงสร้างข้อมูลสำหรับ Sparse Table

- ข้อดีของการแก้ปัญหาแบบ Top-Down ก็คือ เราจะใช้เวลาไปกับช่องข้อมูลที่เกี่ยวข้องกับคำตอบเท่านั้น
- เป็นไปได้ว่าในตารางขนาดใหญ่อาจจะใช้จริง ๆ แค่ไม่กี่ช่อง บางทีแค่ประมาณ 10% ก็เพียงพอ
- แต่เราก็ต้องคอยเก็บผลลัพธ์ไว้หลายระดับ ทำให้ดูเหมือนว่าเราจะต้องเก็บตารางขนาดใหญ่ไว้
- อย่างไรก็ตามแท้จริงแล้ว เราใช้โครงสร้างข้อมูลอย่างอื่นมาช่วยในการจัดเก็บและค้นหาผลลัพธ์แทนตารางแบบเต็มก็ได้
  - เช่นใช้ Tree Map (std::map) หรือ Unordered Map
  - การค้นหาจะช้าลง แต่โดยรวมแล้วถ้าตารางส่วนใหญ่เป็นพื้นที่เปล่า วิธีนี้ถือว่าประหยัดหน่วยความจำและการค้นหาก็ไม่ช้าเกินไปมากนัก
- ที่จริงแบบ bottom-up ก็ทำได้คล้าย ๆ กัน

หัวข้อเนื้อหา

- อะไรคือไดนามิกโปรแกรมมิง (แบบผิวเผิน)
- ประโยชน์ของไดนามิกโปรแกรมมิง
- ตัวอย่างปัญหาที่แสดงแนวคิดและลักษณะเด่นของไดนามิกโปรแกรมมิง
- **อะไรคือไดนามิกโปรแกรมมิง (แบบจริงจัง)**
- จะรู้ได้ไงว่าจะใช้ไดนามิกโปรแกรมมิงได้หรือเปล่า
- กระบวนท่ามาตรฐาน
  - กระบวนท่าในยุคดั้งเดิม (classical techniques)
  - กระบวนท่าในยุคสมัยใหม่ (modern techniques)

อะไรคือไดนามิกโปรแกรม (แบบจริงจัง)

- เป็นการแก้ปัญหาการหาค่าที่ดีที่สุดด้วยการแยกปัญหาใหญ่ออกเป็นปัญหาย่อย โดยที่
  - ปัญหาย่อยมีความซ้ำซ้อนกัน (คือปัญหาใหญ่สองอันมีปัญหาย่อยอันเดียวกัน)
  - คำตอบจากปัญหาย่อยสามารถนำมาใช้คำนวณเป็นคำตอบของปัญหาใหญ่ได้
- การแก้ปัญหาใหญ่จากปัญหาย่อยทำได้จาก The Bellman Equation
  - ➢ สมการมีลักษณะของการนิยามแบบ recursive

$$V(x) = \max_{a \in \Gamma(x)} \{F(x,a) + \beta V(T(x,a))\}$$

อธิบายสมการ Bellman เพิ่มเติม

- จากสมการ

$$V(x) = \max_{a \in \Gamma(x)} \{F(x,a) + \beta V(T(x,a))\}$$

  - x คือ สถานะ (state หรือ ปัญหาย่อย)
  - $V(x)$ คือ ค่าที่ดีที่สุดของสถานะ x (best Value of state x)
  - $\boldsymbol{\Gamma}(x)$ คือ เซตของการกระทำทั้งหมดที่เป็นไปได้ในสถานะ x
  - $T(x,a)$ คือ สถานะที่ได้จากการเลือกทำการกระทำ a (action) (เป็นฟังก์ชันที่คำนวณว่าถ้าทำ a แล้วจะไปต่อที่สถานะใด)
  - $F(x,a)$ คือ ค่าที่เกิดจากการเลือกทำการกระทำ a (เป็นฟังก์ชันคำนวณคุณค่าที่เกิดขึ้นจากการกระทำ a)

Optimization Equation สำหรับปัญหาเลือกชุด

ในทางปฏิบัติเราสามารถเขียนบรรยายการคำนวณปัญหาในรูปแบบรีเคอร์ซีฟได้ และเราจะเขียนบรรยายกรณีพื้นฐานไว้ด้วย

$$V(c,m) = \begin{cases} -1 \text{ ถ้า } m > M \\ m \text{ ถ้า } c = C \\ \max_{a \in K(c)} V(c+1, m+P(a)) \end{cases}$$

- สังเกตได้ว่าสมการตรงส่วนหาค่าสูงสุดในปัญหามันจะดูง่ายกว่ารูปแบบทั่วไปของ Bellman Equation มาก

$$V(x) = \max_{a \in \Gamma(x)} \{F(x,a) + \beta V(T(x,a))\}$$

- คนส่วนใหญ่ในสายวิชาคอมพิวเตอร์จึงมักจะไม่ใช้ตัว Bellman Equation โดยตรงแต่มุ่งความสนใจในการบรรยายปัญหาแบบรีเคอร์ซีฟเลย

แล้วจะรู้ได้ไงว่าปัญหาแก้ได้ด้วยไดนามิกโปรแกรมมิง

- เป็นเรื่องที่ดูออกยากแต่ว่า ...
  - ถ้าเราคิดการค้นหาคำตอบแบบ backtracking ได้และมองเห็นว่ามันมีการคำนวณที่ซ้ำซ้อน (Overlapping sub-problem) แบบนี้มีลุ้นมาก
    → เพราะการใช้ไดนามิกโปรแกรมมิงจะไม่ตัดกรณีใด ๆ ออกไปเลย (แบบนี้มีแนวโน้มจะออกมาในรูป Top-Down แทบจะตรง ๆ)
  - ถ้ามีปัญหาย่อยที่ซ้ำซ้อนและเราเห็นว่าปัญหาใหญ่สามารถคำนวณได้จากปัญหาย่อย แสดงว่ามันมี Optimal sub-structure
    → ใช้ไดนามิกโปรแกรมมิงได้แน่นอน (มีองค์ประกอบสำคัญสองส่วนครบ)
  - ในแบบหลังนี้จะดูออกได้ค่อนข้างยาก เวลาที่เราดูออกเรามักจะมองเห็นเป็นวิธีแบบ Bottom-Up เพราะเราคิดจากปัญหาย่อยก่อนว่าจะเอามาประกอบเป็นปัญหาที่ใหญ่ขึ้นได้อย่างไร

หัวข้อเนื้อหา

- อะไรคือไดนามิกโปรแกรมมิง (แบบผิวเผิน)
- ประโยชน์ของไดนามิกโปรแกรมมิง
- ตัวอย่างปัญหาที่แสดงแนวคิดและลักษณะเด่นของไดนามิกโปรแกรมมิง
- อะไรคือไดนามิกโปรแกรมมิง (แบบจริงจัง)
- จะรู้ได้ไงว่าจะใช้ไดนามิกโปรแกรมมิงได้หรือเปล่า
- **กระบวนท่ามาตรฐาน**
  - กระบวนท่าในยุคดั้งเดิม (classical techniques)
  - กระบวนท่าในยุคสมัยใหม่ (modern techniques)

กระบวนท่าในยุคดั้งเดิม

- 0-1 Knapsack
- ลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด (Longest Increasing Subsequence)
- ลำดับย่อยเหมือนกันที่ยาวที่สุด (Longest Common Subsequence)
- สตริงย่อยเหมือนกันที่ยาวที่สุด (Longest Common Substring)
- Floyd's all-pairs shortest path algorithm
- Planning Company Party
- Word Wrapping (การจัดคำในหน้ากระดาษให้สวยงาม)
- Various Counting Methods
- Range Sum / Maximum (Contiguous) Sum (1D, 2D, ...)

0-1 Knapsack

ปัญหา สมมติว่าเราต้องการเตรียมของใส่เป้สะพาย (Knapsack) สำหรับเดินป่า ปัญหามีอยู่ว่าเรามีของที่อยากใส่เป้หลายอย่างมาก แต่ความจุเป้เราก็ไม่มากนัก เราจึงต้องคำนึงถึงความสำคัญของสิ่งของแต่ละชิ้น เพื่อที่เราจะได้จัดของใส่เป้ได้ดีที่สุด

===

ปัญหานี้จัดว่าง่ายกว่าปัญหาเลือกชุดที่พูดไปในตอนแรก แต่มันถูกเลือกมาใส่ตรงนี้เพื่อให้เราฝึกกับการมองปัญหาในรูปแบบคณิตศาสตร์ที่เป็นทางการมากขึ้น พร้อมกับสังเกตถึง 'ธรรมชาติบางอย่าง' ของการแก้ปัญหาด้วยไดนามิกโปรแกรมมิงที่ชัดเจนขึ้น

ค่าต่าง ๆ ที่เกี่ยวข้องกับ 0-1 Knapsack

- มีวัตถุอยู่ N ชิ้น แต่ละชิ้นมีคุณสมบัติสองอย่างคือ
  - ความจุที่ต้องใช้ในการเก็บลงเป้ (เช่น 600 ลูกบาศก์เซนติเมตร)
  - ระดับความจำเป็น (เช่น ความจำเป็นระดับ 3 หรือระดับ 10)
- ต้องการจัดเป้เพื่อให้ผลรวมระดับความจำเป็นมีค่าสูงสุด
- จำนวนชิ้นที่เก็บลงเป้ได้อาจจะมีตั้งแต่ 0 ชิ้น (เก็บไม่ได้เลย) ไปจนถึง N ชิ้น (เก็บได้ทั้งหมด)
  - แต่ที่แน่ ๆ คือในตอนต้นเราไม่รู้และไม่ได้มีข้อบ่งชี้เป็นพิเศษว่าเก็บกี่ชิ้นถึงจะได้ผลลัพธ์ที่ดีที่สุด
  - เป็นไปได้ว่าเก็บ A ชิ้น อาจจะให้ผลลัพธ์ที่ดีเทียบเท่ากับเก็บ B ชิ้น โดยที่ A != B

แนวคิด 0-1 Knapsack

- กำหนดให้ของชิ้นที่ i มีมูลค่าความจำเป็นเท่ากับ v_i และความจุที่ใช้คือ c_i
- ทางเลือก (action) ที่เป็นไปได้คือ เลือกหรือไม่เลือกของชิ้นที่ i
  - เมื่อเลือกไปแล้วจะเกิดการเปลี่ยนแปลงกับค่าที่ต้องสนใจอยู่สองอย่างคือ (1) มูลค่ารวม และ (2) ความจุรวมที่ใช้ไป
  - มูลค่ารวม ($\sum v$) และ ความจุรวม ($\sum c$)
- อย่าลืมว่าบางที action ก็มีได้แบบเดียว คือห้ามเลือก เพราะเลือกแล้วเกินความจุเป้
- เราพิจารณาการเลือกสิ่งของจากชิ้นที่ 1 ไปถึงชิ้นที่ N ได้
- สิ่งที่เกี่ยวข้องที่เห็นได้ในตอนนี้ก็คือ การเลือก, ความจุรวม และ มูลค่ารวม
- มูลค่ารวมคือคำตอบ ส่วนการเลือกคือ action และความจุรวมเป็นสถานะ

ดูกันก่อนว่าใช้ไดนามิกโปรแกรมมิงได้จริงหรือเปล่า (1)

ดูที่สถานะน้ำหนักของเป้กับของที่เลือกไปก่อน (อย่าเพิ่งไปพะวงกับมูลค่ารวมที่เป็นตัวเลขคำตอบเอาแค่ที่เกี่ยวกับสถานะและ action ก็พอ)

- สมมติว่าของมีสามชิ้น น้ำหนักเป็น 3, 5 และ 8 ตามลำดับ
- ถ้าเลือกใส่ของสองชิ้นแรกที่มีน้ำหนักเป็น 3 กับ 5 แต่ไม่เลือกชิ้นที่สาม
  → เราได้น้ำหนักรวม 8
- ถ้าไม่เลือกสองชิ้นแรก แต่เลือกใส่ชิ้นที่สามอย่างเดียว
  → เราได้น้ำหนักรวม 8 เหมือนกัน
- แบบนี้แสดงว่าสถานะเป้ของเราเหมือนกันได้แม้ว่าเราจะเลือกทำในสิ่งที่ต่างกันไป → มีปัญหาย่อยที่ซ้อนเหลื่อมกัน (overlapping subproblem)

ดูกันก่อนว่าใช้ไดนามิกโปรแกรมมิงได้จริงหรือเปล่า (2)

จากสถานะที่เหมือนกัน ต้องมาตรวจเพิ่มเติมอีกว่ามูลค่ารวมที่เป็นตัวเลขคำตอบสามารถนำมาจากทางเลือกอันใดอันหนึ่งได้อย่างถูกต้องหรือไม่

- สมมติว่าทางเลือกแรกให้มูลค่ารวมเท่ากับ 5 และทางเลือกที่สองให้มูลค่ารวมเท่ากับ 7
- เราต้องดูกันเลยว่า ทางเลือกแรกมีสิทธิ์ที่จะกลับมาเป็นทางเลือกที่ดีกว่าทางที่สองหรือไม่ ซึ่งเราพบว่ามันจะไม่มีทางกลับมาดีกว่าได้
- ที่ไม่มีทางกลับมาดีกว่าได้ก็เพราะหลังจากนี้ การกระทำใดที่ทางเลือกแรกทำได้ ทางเลือกที่สองก็ทำได้เหมือนกันหมด
→ ทางเลือกที่สองมีมูลค่าที่สูงกว่า ก็จะมีมูลค่าที่สูงกว่าไปเรื่อย ๆ
- แบบนี้แสดงว่ามีโครงสร้างย่อยเหมาะที่สุด (optimal substructure) ด้วย
→ ใช้ไดนามิกโปรแกรมมิงได้

เรื่องที่คนชอบสับสนในไดนามิกโปรแกรมมิง (1)

บางทีเราดูไม่ออกว่ามันใช้ไดนามิกโปรแกรมมิงได้ เพราะเราคิดว่า “เลือกของไปแล้วแบบหนึ่ง แต่มันก็มีทางเลือกอื่นที่อาจจะดีกว่าที่จะมีสิทธิ์กลับมาดีกว่าได้ในท้ายที่สุด” ทำให้ไม่มี optimal substructure

- เช่น ถ้าเลือกของชิ้นแรกแต่ไม่เอาชิ้นที่สองกับสาม เราคิดว่าแบบนี้ถึงมูลค่าจะน้อย แต่น้ำหนักรวมก็น้อย ถ้าหากมีของชิ้นที่สี่ที่มีมูลค่ามากเพิ่มเข้ามา
- แบบนี้วิธีที่ดูมีมูลค่าน้อยกว่าก็มีสิทธิ์กลับมาชนะได้
→ บางคนจะไปคิดไปว่า ใช้ไดนามิกโปรแกรมมิงไม่ได้
- แต่นั่นเป็นความสับสนในกระบวนการคิด เพราะเราเอาของที่อยู่คนละสถานะมาเปรียบเทียบแบบนั้นไม่ได้ กล่าวคือถ้าเราเลือกใส่เฉพาะของชิ้นแรก สถานะน้ำหนักมันคือ 3 ไม่ใช่ 8 จึงเอามาเทียบกันไม่ได้

กำจัดความสับสนแยกคิดเฉพาะสถานะกันก่อน

- ให้เริ่มคิดเฉพาะทางเลือกที่เป็นได้ (action) กับการเปลี่ยนแปลงสภาพข้อจำกัดต่าง ๆ (สถานะ, state)
- เราจะเห็นกันตั้งแต่แรกเลยว่าสถานะอันเดียวกันมาจากทางเลือกที่ต่างกันได้หรือไม่
  - ถ้าไม่ได้ ก็หมดลุ้นใช้ไดนามิกโปรแกรมมิง
  - ถ้าได้ ให้เอาเฉพาะสถานะที่เทียบเท่ากัน มาเปรียบเทียบต่อไปเกี่ยวกับคำตอบสุดท้ายที่จะเกิดตามมา
  - ถ้าทางเลือกสองทางจะเกิดชุดของ action แบบเดียวกันตามมา แบบนี้ปลอดภัยแน่นอน เพราะของที่ดีกว่า ก็จะดีกว่าไปเรื่อย ๆ → สถานะจากสองทางเลือกเทียบเท่ากันจริง
  - แต่ถ้าทางเลือกสองทางมันทำให้เกิดชุดของ action ที่ต่างกัน → อันตราย/สถานะที่คิดไว้ไม่ได้เทียบเท่ากันจริง

นิยามสถานะของปัญหาให้ดี ๆ

- สถานะของปัญหามักจะไม่ถูกกำหนดด้วยตัวแปรเดียว
  - ดูให้ดี ๆ ว่าแม้แต่ในปัญหา 0-1 knapsack **น้ำหนักของสิ่งของในเป้ก็ไม่ใช่ตัวกำหนดสถานะอันเดียว**
  - ที่จริงมี**จำนวนสิ่งของที่เราพิจารณาไปแล้วเป็นส่วนหนึ่งของสถานะ**ด้วย
  - เช่น น้ำหนักรวมมีค่าเท่ากับ 8 แต่พิจารณาไปแค่สองชิ้น กับน้ำหนักรวมเท่ากับ 8 แต่พิจารณาไปแล้วสิบชิ้น *แบบนี้ถือว่าเป็นคนละสถานะ เอามาเทียบกันไม่ได้*
- สถานะของปัญหามักเกี่ยวข้องกับงานที่เราพิจารณาไปแล้ว และบางทีก็เกี่ยวข้องอยู่กับเฉพาะงานที่พิจารณาไปแล้วแค่นั้นด้วย
  - ➢ ปัญหาพวกนี้มักจะดูออกได้ยากขึ้น เช่น โจทย์ Schedule ใน TOI’ 8

สมการสำหรับการคำนวณ 0-1 Knapsack (1)

กำหนดให้ c คือความจุสูงสุดของเป้ และ $V(i,c)$ คือมูลค่ารวมสูงสุดของสิ่งของที่บรรจุในเป้ที่**ใช้ความจุไปแล้วไม่เกิน c** เมื่อเลือกใส่สิ่งของจากชิ้นที่ 1 ถึงชิ้นที่ i

คิดแบบคร่าว ๆ ในตอนแรก (เน้นคิดที่ผลจาก action ที่เป็นไปได้ จากวัตถุน้ำหนัก c_i)

$$V(i,c) = \begin{cases} V(i-1,c) & c_i > c \\ \max(V(i-1,c), V(i-1,c-c_i)+v_i) & c_i \le c \end{cases}$$

- สมการทางด้านบนเกิดจากการที่ของชิ้นที่ i หนักเกินกว่าค่า C
  action ที่เป็นไปได้จึงมีแบบเดียว (คือวัตถุ i ไม่ถูกเลือกใส่ในเป้)
  → มูลค่ารวมก็ต้องคงเดิมจากขั้นตอนที่แล้ว
- สมการทางด้านล่างนั้น เรามีทางเลือกสองทาง ทางแรกคือไม่เอาของชิ้นที่ i ใส่เป้ (ทำให้มูลค่ารวมและความจุที่ใช้ไปคงเดิมจากขั้นที่ผ่านมา) ส่วนทางเลือกที่สองเราเอาของชิ้นที่ i ด้วย ทำให้มูลค่ารวมเพิ่มขึ้นจากเดิมเป็นปริมาณ v_i แต่ความจุที่ต้องใช้ไปก็เพิ่มจากเดิมด้วย
- คำตอบสุดท้ายได้มาจากการหาค่า $V(N,C)$

ความหลากหลายของสมการ

เมื่อเราจะทำสมการให้สมบูรณ์ ...

บางทีสมการที่เราได้จะต่างจากเพื่อนเล็กน้อยตรงวิธีจัดการกรณี base case

อันนี้เข้าใจได้โดยง่าย เช่น หากเราบอกว่าจะเริ่มเติมตารางแถวที่ i = 0 ด้วย

$$V(i,c) = \begin{cases} 0 & i = 0 \\ V(i-1,c) & c_i > c \\ \max(V(i-1,c), V(i-1,c-c_i)+v_i) & c_i \le c \end{cases}$$

แต่ถ้าเราบอกว่าเราจะเริ่มเติมตารางจากแถวที่ i = 1 ขึ้นไปเท่านั้น

$$V(i,c) = \begin{cases} 0 & i = 1, c < c_1 \\ v_1 & i = 1, c \ge c_1 \\ V(i-1,c) & c_i > c \\ \max(V(i-1,c), V(i-1,c-c_i)+v_i) & c_i \le c \end{cases}$$

สมการสำหรับการคำนวณ 0-1 Knapsack (2)

แต่ถ้าเราคิดอีกแบบ คือนิยาม $V(i,c)$ ต่างไปจากเดิม สมการมันจะต่างกันพอสมควร

กำหนดให้ C คือความจุสูงสุดของเป้ และ $V(i,c)$ คือมูลค่ารวมสูงสุดของสิ่งของที่บรรจุในเป้ที่<u>เหลือความจุ **C**</u> เมื่อเลือกใส่สิ่งของจากชิ้นที่ 1 ถึงชิ้นที่ i

คิดแบบคร่าว ๆ ในตอนแรก (เน้นคิดที่ผลจาก action ที่เป็นไปได้)

$$V(i,c) = \begin{cases} V(i-1,c) & c_i > c \\ \max(V(i-1,c), V(i-1,c+c_i)+v_i) & c_i \le c, \\ & c_i + c \le C \end{cases}$$

- สมการแรกเกิดขึ้นมาจากการที่ขั้นตอนที่แล้วมีความจุเหลือไม่ถึง c_i
  → ไม่สามารถใส่ของชิ้นที่ i ได้ มูลค่าและความจุที่เหลือคงเดิม
- สมการที่สองเกิดจากการเลือกใส่ของชิ้นที่ i ได้ ดังนั้นมูลค่ารวมหลังใส่ของชิ้นที่ i จึงเพิ่มขึ้น v_i แต่ความจุที่เหลือก็ลดลงจากขั้นตอนที่แล้วไป c_i
- คำตอบได้มาจากการหาค่า V ที่มากที่สุดเมื่อ $i = N$
- **ทำไมถึงไม่หาคำตอบจาก $V(N,0)$ ไปเลยล่ะ ? (คือหาคำตอบตรงที่เป้เต็มไปเลย)**

เรื่องที่คนชอบสับสนในไดนามิกโปรแกรมมิง (2)

- ได้คำตอบผิด เพราะใช้ความเคยชินว่าค่าที่มุมสุดของตารางคือคำตอบที่ดีที่สุดเสมอ
  - ที่จริงมันขึ้นอยู่กับการนิยามฟังก์ชันผลเฉลย
  - คิดคนละแนวทางก็อาจจะได้วิธีหาคำตอบสุดท้ายที่แตกต่างกันได้
  - ที่จริงเราจะนิยามฟังก์ชันแนวทางไหนก็ได้ ขอแค่นำไปใช้ให้ถูกต้องก็พอ
- ลองดูกรณีที่ $c_i = \{1,2\}$, $v_i = \{3,1\}$ และ $C = 2$

V(i, c)	c = 0	c = 1	c = 2
i = 1	N/A	3	0
i = 2	1	3	0

- เห็นได้ว่า $V(2,0)$ ไม่ใช่คำตอบที่ดีที่สุด เพราะการที่ความจุไม่เหลือเลย ไม่ได้เป็นหลักประกันว่าจะให้มูลค่ารวมที่ดีที่สุด

เปรียบเทียบกับตอนใช้ฟังก์ชัน V(i, c) แบบเดิม

- ถ้าให้ c แทนขอบเขตความจุสูงสุดที่ใช้ไป
  → การใช้ขอบเขตความจุจะให้ความหมายและการใช้งานที่ครอบคลุมกว่า

V(i, c)	c = 0	c = 1	c = 2
i = 1	0	3	3
i = 2	0	3	3

- เป็นวิธีที่หนังสือส่วนใหญ่เลือก
- แต่ธรรมชาติในการคิดของแต่ละคนก็ไม่เหมือนกัน ยากมากที่จะบอกให้ทุกคนคิดเหมือนกันหมด
- เอาเป็นว่า ไม่ต้องกังวลว่าจะต้องเหมือนเพื่อน แต่ขอให้รู้ด้วยว่าวิธีคิดที่แตกต่างก็ถูกทั้งคู่ได้ แต่ต้องหยิบคำตอบสุดท้ายให้สอดคล้องกับวิธีตัวเอง

ธรรมชาติสำคัญบางอย่างในไดนามิกโปรแกรมมิง

- มักมีการเลือกตัดสินใจว่า ทำ-ไม่ทำ หยิบ-ไม่หยิบของชิ้นหนึ่ง หรือ เลือกของชิ้นเดียวจากตัวเลือกที่มีให้
- เลือกไปแล้วปัญหาก็ดูคล้าย ๆ เดิม คือต้องทำอะไรเหมือน ๆ เดิมไปเรื่อย ๆ จนกว่าจะถึงจุดหมาย
- การกระทำอันดับต่อไปที่จะกระทำได้มักผูกอยู่กับสถานะของปัญหา
- ถ้าทางเลือกสองทางที่ต่างกันทำแล้วนำไปสู่เซตของการกระทำที่เหมือนกันตลอดจนถึงปลายทางของปัญหา แสดงว่ามันมีปัญหาย่อยที่คาบเกี่ยวกัน
  - เป็นมุมมองของการวิเคราะห์ปัญหาที่ช่วยเราได้มาก
- ถ้าทางเลือกที่แย่กว่าของปัญหาในสถานะเดียวกัน จะไม่มีทางกลับมาเอาชนะทางเลือกอีกทางได้ แสดงว่ามันมีโครงสร้างย่อยเหมาะที่สุด

ข้อสังเกตพิเศษในบางปัญหา

- ประเด็นหลักอย่างหนึ่งในไดนามิกโปรแกรมมิงก็คือ การแทนที่ทางเลือกที่ 'สิ้นหวัง' (ทางเลือกที่แย่กว่า) ด้วยทางเลือกที่ 'ยังมีความหวัง' (ทางเลือกที่ดีกว่า)
- วิธีการใดก็ตามที่ตัดตัวเลือกที่สิ้นหวังได้เร็วก็จะมีความโดดเด่นเรื่องประสิทธิภาพขึ้นมาทันที
  - บางวิธีจะถูกเปลี่ยนไปเป็นมุมมองของ Greedy Algorithm
  - บางวิธีจะยังเป็นมุมมองไดนามิกโปรแกรมมิงอยู่เช่นเดิม (แต่ดูออกยาก มักจะต้องมีคนสอนปูพื้นให้ถึงมองออก)
- ตัวอย่างแบบที่ยังเป็นไดนามิกโปรแกรมมิง: Longest Increasing Subsequence และ Maximum Sum (Kanade's Algorithm)

กระบวนท่าในยุคดั้งเดิม

- 0-1 Knapsack
- **ลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด (Longest Increasing Subsequence)**
- **ลำดับย่อยเหมือนกันที่ยาวที่สุด (Longest Common Subsequence)**
- **สตริงย่อยเหมือนกันที่ยาวที่สุด (Longest Common Substring)**
- Floyd's all-pairs shortest path algorithm
- Planning Company Party
- Word Wrapping (การจัดคำในหน้ากระดาษให้สวยงาม)
- Various Counting Methods
- Range Sum / Maximum (Contiguous) Sum (1D, 2D, ...)

ลำดับย่อยเพิ่มขึ้น (Increasing Subsequence)

- ลำดับ (sequence) คือ ชุดของเหตุการณ์ซึ่งการปรากฏก่อนหลังมีความสำคัญ
  - เช่น ถ้าเหตุการณ์ในลำดับคือ a, b, c การสลับตำแหน่งเป็น b, a, c จะมีความหมายเป็นลำดับคนละอย่างกัน
  - โดยมากเราแทนเหตุการณ์เป็นตัวเลข เช่น 7, 3, 4, 5, 2, 1, 6
  - ลำดับย่อยคือชุดของเหตุการณ์จากลำดับเต็มที่ไม่มีการสลับการปรากฏก่อนหลัง เช่น 3, 5, 2, 1, 6 หรือ 7, 3, 4, 2 หรือ 3, 4, 6 เป็นต้น
  - นั่นคือเหตุการณ์จากลำดับเต็มจะติดกันหรือไม่ติดกันก็ได้
  - ลำดับย่อยเพิ่มขึ้นคือ ลำดับย่อยที่เหตุการณ์ที่มาก่อนเป็นตัวเลขที่มีค่าน้อยกว่าเหตุการณ์ที่มาทีหลัง เช่น 3, 4, 6 จัดเป็นลำดับย่อยเพิ่มขึ้น ในขณะที่ 3, 5, 2 และ 7, 3, 4 ไม่ใช่ลำดับย่อยเพิ่มขึ้น

ลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด (1)

- คือลำดับย่อยเพิ่มขึ้นที่มีจำนวนเหตุการณ์ภายในมากที่สุด
  - ➢ อาจจะเป็นตัวเดียวกันกับลำดับเต็มก็ได้
- มีปัญหาย่อยที่ซ้อนเหลื่อมกันหรือไม่
  - เราสังเกตได้ว่าการเลือกหรือไม่เลือกตัวเลขเข้ามาในลำดับส่งผลกับการกระทำที่เหลือ
  - เฉพาะตัวเลขล่าสุดที่ส่งผลกับทางเลือกที่เหลือทั้งหมดที่เป็นไปได้ ถ้าตัวเลือกล่าสุดเหมือนกัน ทางเลือกที่เหลือที่เป็นไปได้ก็เหมือนกัน
  - เช่น ถ้าลำดับเต็มคือ 7, 3, 4, 5, 2, 1, 6 และตัวสุดท้ายในลำดับย่อยคือ 5 เราจะพบว่าก่อนหน้านี้เราจะเลือก 3 หรือ 3, 4 หรือ 4 มันก็จะไม่ส่งผลต่อทางเลือกที่เหลือเลย คือตัวสุดท้ายเท่านี้ที่ส่งผลต่อ action ที่จะตามมา

ลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด (2)

- แล้วมีโครงสร้างย่อยเหมาะที่สุดหรือไม่
  - พิจารณาลำดับเต็ม 4, 8, 0, **12**, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15
  - หากเราเลือกที่จะใส่เลข 12 เข้ามาในลำดับ เราก็สามารถเลือกให้มันต่อกับลำดับย่อย 4, 8 หรือลำดับย่อย 0 ก็ได้
  - ลำดับย่อย 0 มีเลขแค่ตัวเดียวทำให้ความยาวหลังรวมเลข 12 คือ 2 ตัว
  - ในขณะที่การต่อเข้าที่ 4, 8 จะได้ความยาวหลังรวมเลขคือ 3 ตัว
  - เนื่องจากการกระทำที่เหลือของทั้งสองทางเลือกเหมือนกันหมด ทางเลือกที่ได้ลำดับย่อยที่ยาวที่สุดเท่านั้นที่ยังมีความหวัง
    ทางเลือกที่ไม่ได้ให้ลำดับย่อยที่ยาวที่สุดหมดหวังแล้วแน่นอน
- ดังนั้นปัญหานี้มีโครงสร้างย่อยเหมาะที่สุดด้วย

วิธีคำนวณแบบพื้น ๆ

ลองพิจารณาตัวเลขตัวหนึ่งจากซ้ายไปขวา

- การกระทำที่ทำได้มีอยู่สองกลุ่มคือ เลือกรวมเลขตัวนั้นเข้ามาในลำดับย่อย หรือไม่รวมเลขตัวนั้นเข้ามาในลำดับย่อย
- ในกรณีที่เลือกรวมเข้ามาก็มีทางเลือกเหมือนกันว่าจะต่อท้ายเข้ากับเลขใด
- แน่นอนว่าเราต้องเลือกต่อเข้ากับลำดับย่อยที่มีความยาวมากที่สุด (ถ้าต่อได้) แต่ถ้าต่อกับใครไม่ได้เลยก็คือใช้เลขตัวนั้นเป็นตัวเริ่มต้นเลย
- เช่น ในตัวอย่างเดิมเราไม่สามารถเลือกเลข 0 ไปต่อท้ายใครได้
  → ต้องเริ่มลำดับย่อยใหม่ด้วยเลข 0
- แล้วถ้าเราบอกว่าจะไม่เลือกตัวเลขที่พิจารณาอยู่เข้ามาในลำดับล่ะ ?
- การกระทำที่เหลือและผลลัพธ์เหมือนกับผลที่เราทำไว้กับเลขก่อนหน้า
  → ปัญหาย่อยจะซ้ำกัน ไม่จำเป็นต้องคิดซ้ำซ้อนอีก

ทางเลือกในการต่อลำดับ

- ทางเลือกในการต่อลำดับที่เป็นไปได้ทั้งหมดก็คือต่อเลขเข้ากับลำดับย่อยที่เลขด้านท้ายมีค่าน้อยกว่าหรือเท่ากับเลขที่พิจารณาอยู่ทั้งหมด
  - เลขลำดับที่ i จะมีเลขก่อนหน้าที่เราจะทดลองต่อลำดับทั้งหมด i – 1 ตัว
  - การพิจารณาเลขแต่ละตัวจึงต้องใช้เวลาเป็น $O(i)$
  - ถ้าทำจนเสร็จเวลาก็จะเป็น
    $$O(0+1+2+\cdots+N-1)=O(N^2)$$
  - เรื่องน่าสังเกตจากตัวอย่างเดิมก็คือว่า ตอนที่เราจะต่อเลข 12 เข้ากับลำดับที่มีเลข 0 แค่ตัวเดียว นับว่าเป็นทางเลือกที่ไร้ความหมาย เพราะ 0 ตัวนั้นเป็นส่วนหนึ่งของลำดับย่อยที่ดีกว่า คือลำดับ 0, 8 ภายใต้การต่อลำดับด้วย 12
  - แต่เราก็ต้องเก็บ 0 ไว้ เพราะหากเลขที่พิจารณาเป็น 7 เราจะต่อท้าย 8 ไม่ได้
- **มีวิธีที่ที่ทำให้เราสามารถคัดทางเลือกที่ไม่มีประโยชน์ออกไปเร็ว ๆ หรือไม่ ?**

ข้อสังเกตพิเศษสำหรับปัญหาลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด

- ทางเลือกที่ไม่มีประโยชน์ในปัญหานี้คือ
  1. ต่อตัวเลขที่พิจาณาเข้าไปไม่ได้ เพราะขัดกับข้อกำหนดการเพิ่มขึ้นของลำดับ
  2. เลือกต่อเข้าไปกลางลำดับย่อยอื่น แทนที่จะเป็นการต่อเข้าที่ตัวปลายสุดที่เลขตัวนั้นสามารถต่อได้
     - กรณีนี้ก็คือการต่อเลข 12 เข้าที่เลข **4** ทั้งที่ตัวปลายสุดที่ต่อได้คือ 0, 8
     - ดูอีกตัวอย่าง หากลำดับเต็มคือ 1, 2, 3, 15, 12 และเรากำลังพิจารณาเลข 12 อยู่ เราพบว่าลำดับย่อยที่ยาวสุดก่อนหน้าคือ 1, 2, 3, 15
     - แต่เลขปลายสุดที่ต่อเข้าไปได้คือเลข 3 ดังนั้นเลขในลำดับเดียวกันตัวก่อนหน้าทั้งหมดคือ 1 และ 2 ล้วนจัดเป็นทางเลือกที่ไม่มีประโยชน์ทั้งนั้น

กำจัดข้อมูลส่วนเกินที่ไม่มีประโยชน์

- การเก็บลำดับข้อมูลของลำดับย่อยเพิ่มขึ้นสองชุดที่ยาวเท่ากันไว้ก็ไม่มีประโยชน์ เช่น ถ้าเรามีลำดับย่อย 13, 14, 15, 1, 2, 3
  - ลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด ณ ปัจจุบันมีสองชุดคือ 13, 14, 15 และ 1, 2, 3
  - หากเรามีเลขเพิ่มขึ้นมาในลำดับ เช่น มีเลข 20 มาต่อท้าย เราก็จะพบว่าจะต่อเข้าลำดับย่อยใดก็ให้ผลเหมือนกัน
  - แต่ถ้าเลขที่เพิ่มขึ้นมาเป็น 5 เราจะพบว่า มันต่อเข้าไปที่ 1, 2, 3 ได้แค่ลำดับย่อยเดียว
  - **ดังนั้นถ้ามีลำดับย่อยเพิ่มขึ้นสองตัวที่ยาวเท่ากัน เราเลือกเฉพาะลำดับย่อยที่เลขปิดท้ายมันมีค่าน้อยกว่าก็พอ**
- จากข้อสังเกตพิเศษ เรามองเห็นสิ่งที่ไม่มีประโยชน์ คือเป็นภาระในการคำนวณหรือการจัดเก็บมามากพอสมควรแล้ว
- แล้วทำไงดีเราถึงจะ (1) หลบตัวเลขที่ต่อเข้าไปไม่ได้อย่างรวดเร็ว (2) หาตำแหน่งด้านปลายลำดับย่อยที่ต่อได้อย่างรวดเร็ว

จัดโครงสร้างเพื่อให้ค้นหาตำแหน่งได้ง่าย

- จากการสังเกตเห็นข้อมูลส่วนเกินที่ไม่มีประโยชน์ เราสรุปได้ว่า
  - สำหรับลำดับย่อยความยาว j เราเก็บแค่ตัวเลขท้ายลำดับค่าน้อยที่สุดไว้ก็พอ
  - ดังนั้นในหมู่เลขที่เก็บไว้ ค่าตัวเลขที่มากกว่าจะอยู่ในลำดับย่อยที่มีความยาวมากกว่าด้วย
  - ค่าความยาว j นี้จะเริ่มจาก 1 ไป 2 ไป 3 … ไม่มีการข้าม
- แบบนี้เราใช้ j เป็นดัชนีในอาเรย์ได้ และอาเรย์นี้จะได้รับการจัดลำดับเรียงจากน้อยไปมากด้วย
  - แต่เราก็ต้องคอยปรับค่าตัวเลขในอาเรย์นี้ ซึ่งมีแนวโน้มดังนี้
    - ในการปรับค่าช่องที่ j หนึ่ง ๆ จะถูกเปลี่ยนเป็นตัวเลขที่น้อยลงเรื่อย ๆ
    - ค่า j สูงสุดจะค่อย ๆ เพิ่ม อาเรย์ก็จะยาวขึ้นด้วย แต่ก็ไม่เกิน N แน่นอน

ต้องการค้นคืนลำดับที่เป็นคำตอบ

- มาถึงจุดนี้เราอาจจะเริ่มรู้สึกตัวแล้วว่า
  - เรารู้ความยาวลำดับย่อยเพิ่มขึ้นยาวที่สุด แต่เราไม่รู้ว่าลำดับนั้นมีเลขใดบ้าง
  - เพราะเราสนใจแต่ค่าตัวเลขที่อยู่ปลายสุด เราไม่มีข้อมูลเลยว่าแท้จริงเลขดังกล่าวอยู่ ณ ตำแหน่งใด และเราไม่รู้ด้วยว่าเลขตัวถัดไปมีค่าเท่าใด
- ต้องเก็บข้อมูลด้านตำแหน่งไว้ด้วย
  - ควรเก็บว่าเลขแต่ละตัวต่อเข้ากับเลขที่ตำแหน่งใด
  - เพื่อให้การเชื่อมต่อจากอาเรย์ที่ใช้ในการค้นหา เราจึงเปลี่ยนจากเก็บค่าตัวเลขตรง ๆ ในอาเรย์นั้น แล้วไปเก็บค่าตำแหน่งของตัวเลขดังกล่าวแทน
  - ตำแหน่งที่ว่าคือดัชนีในอาเรย์อินพุต เช่น ถ้าลำดับเต็มคือ 13, 1, 2, 14, 3, 15 อาเรย์จะเก็บตำแหน่งเป็น 2, 3, 5, 6 (ตำแหน่งเริ่มนับจากหนึ่ง)

วิธีนำตำแหน่งไปใช้ในการค้นคืนลำดับย่อย (1)

- ตอนที่เราพิจารณาเลขแต่ละตัวว่าจะไปต่อลำดับที่ไหนดี
  - ให้เราจำตำแหน่งที่มันต่อไว้ด้วย
  - ดังนั้นตอนนี้เรามีสามอาเรย์แล้วคือ
    - อาเรย์ D (Data Array) เก็บข้อมูลเข้า (ลำดับเต็ม) เอาไว้
    - อาเรย์ M (Max Array) เก็บตำแหน่งปลายของลำดับย่อยไว้ (ตำแหน่งใน D ที่เก็บค่าตัวเลขของปลายลำดับย่อยเอาไว้)
    - อาเรย์ P (Position Array) เก็บตำแหน่งการเชื่อมต่อที่ดีที่สุดของเลขแต่ละตัวไว้ (เก็บเฉพาะตำแหน่งที่เชื่อมต่อไปเลขในลำดับย่อยเพิ่มขึ้นที่ติดกับมันและมีค่าน้อยกว่า)

วิธีนำตำแหน่งไปใช้ในการค้นคืนลำดับย่อย (2)

- ตอนค้นคืนลำดับย่อย เราเริ่มจากข้อมูลด้านท้ายสุดในอาเรย์ M
  - ซึ่งก็คือตำแหน่งปลายของลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด
- ดึงค่าจากอาเรย์ D มาเก็บไว้ก่อน (พิมพ์ทันทีไม่ได้ ต้องพิมพ์ย้อนหลัง)
- ดึงค่าจากอาเรย์ P มาเพื่อหาตำแหน่งของตัวเลขถัดไปในลำดับย่อย
- ทำลักษณะเดิมไปเรื่อย ๆ จนกว่าจะถึงต้นลำดับย่อย

ตัวอย่าง

```
D = { 2, 8, 4, 12, 3, 10, 6, 14}
V = { 2, 3, 6, 14}
M = { 1, 5, 7, 8}
P = {-1, 1, 1,  3, 1,  5, 5,  7}
```


ข้อสังเกตพิเศษกับ Dijkstra's Algorithm

ปัญหา Single Source Shortest Path แท้จริงมองเป็นแบบไดนามิกโปรแกรมมิงได้

- จากโหนดเริ่มต้น เรามี action ให้เลือกเส้นทางไปโหนดที่ติดกัน
- เราเลือกอันที่มันให้ค่าระยะทางรวมน้อยที่สุด
- เซตของ action ที่ให้เลือกอาจจะขยายเพิ่มจากโหนดที่รวมเข้ามาใหม่
  - เราพิจารณาเซตดังกล่าวแล้วเลือกเส้นทางที่สั้นที่สุดที่มีอยู่ในนั้น
  - ถ้าเราอัดแบบตรง ๆ เราสามารถสร้างลิสต์ของ action แล้วหาเส้นทางที่สั้นที่สุดจาก action ภายในลิสต์
  - แต่ Dijkstra's Algorithm สามารถดึงเอาค่าระยะทางที่น้อยที่สุดออกมาได้แบบครั้งเดียวเสร็จผ่านการใช้ Priority Queue ทำให้มุมมองของวิธีการถูกพิจารณาว่าเป็น Greedy Algorithm แทนที่จะเป็นไดนามิกโปรแกรมมิง

ปัญหา Longest Common Subsequence (LCS)

เป็นปัญหาคลาสสิคที่มีการประยุกต์ใช้ในปัญหาสมัยใหม่อย่างแพร่หลาย โดยเฉพาะเรื่องของการวิเคราะห์ DNA เช่น นักชีววิทยาอาจจะต้องหาลำดับย่อยที่เหมือนกันที่ยาวที่สุดของสตริง DNA สองชุด

```
S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA
```

ผลจากการเปรียบเทียบจะนำไปสู่การประเมินว่า DNA สองชุดนี้มีความคล้ายกันมากเพียงใด (เป็นงานกลุ่ม Bioinformatics)

ปัญหานี้ถือว่าค่อนข้างยากที่จะสังเกตเห็นคุณสมบัติสำคัญในการแก้ปัญหาได้ด้วยตัวเอง

LCS: คุณสมบัติที่สำคัญ

กำหนดให้ลำดับ $X = < x_1, x_2, \dots, x_m >$ และ $Y = < y_1, y_2, \dots, y_n >$ และกำหนดให้ $Z = < z_1, z_2, \dots, z_k >$ เป็น LCS ของ X และ Y

1. ถ้า $x_m = y_n$ จะได้ว่า $z_k = x_m = y_n$ และ Z_{k-1} เป็น LCS ของ X_{m-1} และ Y_{n-1}
2. ถ้า $x_m \neq y_n$ และ $z_k \neq x_m$ จะได้ว่า Z เป็น LCS ของ X_{m-1} และ Y
3. ถ้า $x_m \neq y_n$ และ $z_k \neq y_n$ จะได้ว่า Z เป็น LCS ของ X และ Y_{n-1}

LCS ขั้นที่ 1: ตรวจดู Action หาปัญหาย่อยซ้อนเหลื่อม

สมมติให้อินพุตเป็น X = <1, 0, 0, 1, 0, 1, 1, 1> และ
Y = <0, 1, 0, 1, 1, 0, 1, 0, 0>

- เนื่องจากตัวสุดท้ายไม่เหมือนกัน การหา LCS ของ X กับ Y จึงเหมือนกับการหา LCS ของ X กับ <0, 1, 0, 1, 1, 0, 1, 0> หรือ LCS ของ Y กับ <1, 0, 0, 1, 0, 1, 1> ณ ตรงนี้เราจะพบว่า ปัญหายังไม่ซ้อนเหลื่อมกันให้เห็น
- จาก LCS ของ X กับ <0, 1, 0, 1, 1, 0, 1, 0> ตัวสุดท้ายก็ยังไม่เหมือนกัน คำตอบจึงเทียบเท่ากับการหา LCS ของ X กับ <0, 1, 0, 1, 1, 0, 1> หรือ LCS ของ **<1, 0, 0, 1, 0, 1, 1> กับ <0, 1, 0, 1, 1, 0, 1, 0>**
- ในทำนองเดียวกัน จาก LCS ของ Y กับ <1, 0, 0, 1, 0, 1, 1> เราจะได้ปัญหาย่อยเป็น LCS ของ Y กับ <1, 0, 0, 1, 0, 1> หรือ LCS ของ **<0, 1, 0, 1, 1, 0, 1, 0> กับ <1, 0, 0, 1, 0, 1, 1>** (มีการซ้อนเหลื่อมกันแล้ว)

LCS: Memo Table

- จากตัวอย่างข้อมูลและวิธีย่อยปัญหา
  - เราลดทอนขนาดของปัญหาด้วยการทำให้ลำดับอินพุตสั้นลงจากด้านท้าย
  - บางทีเราก็ตัดให้ลำดับจากฝั่ง X ให้สั้นลง แต่บางทีเราก็ตัดทางฝั่ง Y และการตัดหลาย ๆ ครั้งก็อาจจะไปบรรจบเป็นปัญหาย่อยอันเดียวกัน
  - เราทดสอบว่าปัญหาย่อยสองอันเป็นปัญหาอันเดียวกันหรือไม่ด้วยการตรวจความยาวของลำดับจากฝั่ง X และ Y
    - กำหนดให้ความยาวลำดับฝั่ง X คือ i ส่วนของฝั่ง Y คือ j
    - ปัญหาสองอันจะเป็นอันเดียวกันถ้าความยาวของลำดับจากฝั่ง X และ Y ในทั้งสองปัญหาเหมือนกันคือเท่ากับ i และ j เหมือนกัน
- ดังนั้นตารางควรจะขึ้นอยู่กับความยาวของ X และ Y และข้างในบรรจุความยาวของ LCS ที่ได้

LCS ขั้นที่ 2: สมการหาค่าที่ดีที่สุด

กำหนดให้ตาราง c เก็บความยาวของ LCS เมื่อลำดับของฝั่ง X ยาว i และลำดับของฝั่ง Y ยาว j เราจะได้สมการหาค่าสูงสุดเป็น

$$c[i,j] = \begin{cases} 0; & \text{ถ้า } i = 0 \text{ หรือ } j = 0 \\ c[i-1,j-1]+1; & \text{ถ้า } i,j > 0 \text{ และ } x_i = y_j \\ \max(c[i,j-1], c[i-1,j])\,; & \text{ถ้า } i,j > 0 \text{ และ } x_i \neq y_j \end{cases}$$

สมการมีความหมายดังนี้

1. $c[i-1,j-1]+1$ มาจากคุณสมบัติที่เรากล่าวถึงไปก่อนหน้าคือลำดับที่พิจารณาทั้งสองลงท้ายด้วยเลขหรือตัวอักษรเดียวกัน
2. $\max(c[i,j-1], c[i-1,j])$ มาจากเหตุการณ์ที่ตัวสุดท้ายในลำดับต่างกัน แต่เรายังไม่รู้ว่าตัดท้ายของลำดับไหนไปถึงจะดีที่สุด เราจึงต้องลองตัดทั้งสองทางแล้วเลือกค่าที่ดีที่สุด

LCS กับปัญหาการสร้างพาลินโดรม (IOI 2000 PALIN)

จงหาว่าจากสตริงความยาว N ที่กำหนดให้ เราสามารถสร้างพาลินโดรมโดยใส่ตัวอักษรเพิ่มเข้าไปน้อยที่สุดกี่ตัว ที่ต้น, ปลาย หรือระหว่างสตริงที่กำหนดให้ ทั้งนี้ตักอักษรใหญ่เล็กถือว่าเป็นคนละตัวอักษรกัน

ตัวอย่าง

ข้อมูลเข้า	ผลลัพธ์
5 Ab3bd	2
4 1231	1
4 1234	3

ข้อนี้จะทำได้ทันต้องประหยัดหน่วยความจำด้วยเพราะใช้เวลา init อาเรย์แบบเต็มนานมาก

สตริงย่อยเหมือนกันที่ยาวที่สุด

- ต่างกับปัญหาลำดับย่อยตรงที่ว่าสตริงย่อยจะต้องเป็นตัวอักษรที่ติดกัน
- ใช้ suffix tree อาจจะเร็วกว่าถ้าจำนวนตัวอักษรในภาษามีไม่มากนัก
- ตัวอย่าง “ABAB” และ “BABAA” มีสตริงย่อยเหมือนกันที่ยาวที่สุดสองแบบคือ “ABA” และ “BAB”
- การคำนวณมันจะดูง่ายขึ้นเพราะเลือกจุดเริ่มต้นของสตริงย่อยจากสตริงทั้งสองมาทดสอบกันเลย ถ้าพบจุดที่มันต่างกันความยาวก็จะกลายเป็นศูนย์เหมือนเริ่มต้นใหม่ แต่ของลำดับย่อยความยาวอาจจะต่อกันไปได้
  - เรื่องมันง่ายตรงที่ว่าสมมติเราจับคู่ A ตัวแรกจาก ABAB ประกบกับ A ตัวแรกใน BABAA ในระหว่างทางการพยายามจับคู่ต่อเนื่องไปเป็น AB เราจะพบว่า ...
  - การเลือก B ตรงนี้จะทำให้ action ที่ตามมาเหมือนกันหมดทั้งในการเลือก B เป็นตัวเริ่มหรือเลือก B เป็นตัวที่สองแบบที่เป็นอยู่ใน AB แสดงว่าปัญหาซ้อนเหลื่อมกัน

ยืนยันการใช้ไดนามิกโปรแกรมมิง

- เรื่องต่อเนื่องก็คือว่า ถ้าสตริงย่อยที่ทำมาก่อนหน้ามีความยาวที่มากกว่า (ซึ่งมันก็เป็นอย่างนั้นโดยธรรมชาติอยู่แล้ว)
  - มันจะยาวกว่าแบบเริ่มใหม่แน่นอน
  - ไม่มีทางเป็นอย่างอื่น ดังนั้นมีโครงสร้างย่อยเหมาะที่สุด
- สมการสำหรับการหาค่าความยาวที่ดีที่สุด
  - ตั้งสมการหาสตริงย่อยเหมือนกันที่ยาวที่สุดเมื่อสตริงย่อยที่เหมือนกันของสตริงที่ S และ T จบที่ตำแหน่งที่ p และ q ตามลำดับ

$$LCStr(S[1..p], T[1..q]) = \begin{cases} 0 & S[p] \neq T[q] \\ LCStr(S[1..p-1], T[1..q-1]) + 1 & S[p] = T[q] \end{cases}$$

ตัวอย่างการเติมค่าลงในตาราง

“ABAB” และ “BABAA”

	B	A	B	A	A
A	0	1	0	1	0
B	1	0	2	0	0
A	0	2	0	3	1
B	1	0	3	0	0

- ค่าจะเติบโตในมุมทแยงเสมอ
- เช่นเดียวกับ LCS เราประหยัดหน่วยความจำได้เพราะมีข้อมูลสองแถวก็พอ
- การประหยัดหน่วยความจำบนตารางแบบนี้สำคัญมากเพราะมันประหยัดเวลาด้วย

กระบวนท่าในยุคดั้งเดิม

- 0-1 Knapsack
- ลำดับย่อยเพิ่มขึ้นที่ยาวที่สุด (Longest Increasing Subsequence)
- ลำดับย่อยเหมือนกันที่ยาวที่สุด (Longest Common Subsequence)
- สตริงย่อยเหมือนกันที่ยาวที่สุด (Longest Common Substring)
- ~~Floyd's all-pairs shortest path algorithm~~
- Planning Company Party
- Word Wrapping (การจัดคำในหน้ากระดาษให้สวยงาม)
- **Various Counting Methods**
- **Range Sum** / Maximum (Contiguous) Sum (1D, 2D, …)

ผลบวกของจำนวนเฉพาะที่แตกต่างกัน

จำนวนเต็มบวกสามารถที่จะถูกเขียนอยู่ในรูปของผลบวกของจำนวนเฉพาะที่ไม่ซ้ำกัน และบางจำนวนก็เขียนได้มากกว่าหนึ่งแบบ ในปัญหานี้ เราต้องการนับวิธีการเขียนแสดงจำนวนเต็ม N ทั้งหมด ในรูปของผลบวกจำนวนเฉพาะ k ตัว โดยจำนวนเฉพาะนี้ไม่มีค่าใดซ้ำกัน และเราถือว่าการสลับลำดับการบวกเป็นวิธีเดียวกัน เช่น 8 = 5 + 3 และ 8 = 3 + 5 ถือเป็นวิธีเดียวกัน

ตัวอย่าง

ถ้า N = 24 และ k = 3 จะมีอยู่สองวิธีคือ 2 + 3 + 19 และ 2 + 5 + 17
ดังนั้นคำตอบคือ 2
ถ้า N = 24 และ k = 2 จะมีอยู่สามวิธีคือ 5 + 19, 7 + 17 และ 11 + 13
ดังนั้นคำตอบคือ 3
แต่ถ้า N = 3 และ k = 2 หรือ N = 1 และ k = 1 คำตอบคือศูนย์

กำหนดให้ $N \leq 1120$ และ $k \leq 14$

กระบวนท่าในยุคสมัยใหม่

ส่วนใหญ่เป็นการประยุกต์ใช้กระบวนท่าในยุคดั้งเดิมในรูปแบบที่มองออกไม่ง่ายนักว่าจะหยิบเอากระบวนท่าในยุคดั้งเดิมมาใช้อย่างไร

- Live-wire Algorithm
- **Saliency Detection**
- Seam Carving (ใช้ไดนามิกโปรแกรมมิงสองตัวคู่กัน ผลลัพธ์สวยมาก)
- Stereo Algorithm for Solving Correspondence Problem
- Beat Tracking

Saliency Detection

- ต้องการตัดพื้นที่ภาพขนาด M x N ให้มีวัตถุเด่น ๆ อยู่มากที่สุด
- โดยพื้นฐานก็คือการทำ 2D Range Sum เพื่อหาพื้นที่ขนาดตายตัวที่มีวัตถุเด่นอยู่มากที่สุด

Image Credit: ภาพยนตร์เรื่อง Indiana Jones
และการตัดภาพอัตโนมัติจาก
http://madlab.cpe.ku.ac.th/autocrop/questionare.php

คำว่ามีวัตถุเด่นมากที่สุดคืออะไร

- นิยามของการมีวัตถุเด่นอาจจะซับซ้อนพอสมควร ทำให้เราไม่สามารถใช้ Range Sum ตรง ๆ ได้ แต่ต้องอาศัยอัลกอริทึมอื่นเข้าร่วมด้วย
- ปัญหามันจะซับซ้อนหนักขึ้นไปอีกในกรณีที่วัตถุที่สนใจอยู่ห่างกัน แต่เราเลือกได้แค่พื้นที่เดียว

Auto-Cropping Results:
http://madlab.cpe.ku.ac.th/autocrop/questionare.php

แล้วเราอยากได้แบบไหน

แบบแบ่งปันความสุข

แบบเอาให้จะแจ้งเท่านั้น

Auto-Cropping Results:
http://madlab.cpe.ku.ac.th/autocrop/questionare.php

สมมติว่าเราเอาแบบจะแจ้ง

สมมติว่า 1 คือช่องที่มีวัตถุที่สนใจ ซึ่งความต่อเนื่องเป็นแบบสี่ทิศทาง ในภาพนี้มีวัตถุอยู่ 3 วัตถุ เราต้องการทำให้พื้นที่ MxN มีเลข 1 ให้มากที่สุด แต่หากจำนวนเลข 1 ของวัตถุในพื้นที่มีน้อยกว่า 50% เราจะไม่นับเลข 1 จากวัตถุนั้น

					1	1	1	1				
	1	1			1	1	1	1		1	1	1
1	1	1	1		1	1	1	1		1	1	1
1	1	1	1		1	1	1	1		1	1	1
1	1	1	1		1	1	1	1		1	1	1
1	1	1	1		1	1	1	1		1	1	1
1	1	1	1			1	1			1	1	1
	1	1				1	1		1	1	1	1

กระบวนท่าในยุคสมัยใหม่

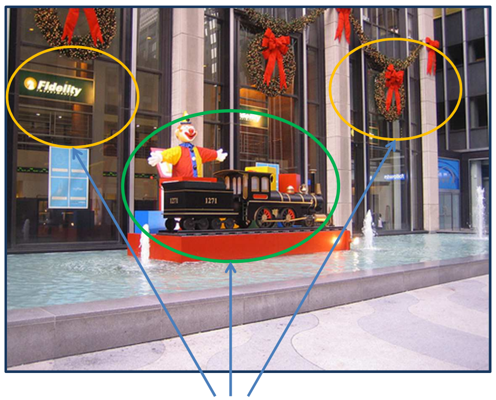
ส่วนใหญ่เป็นการประยุกต์ใช้กระบวนท่าในยุคดั้งเดิมในรูปแบบที่มองออกไม่ง่ายนักว่าจะหยิบเอากระบวนท่าในยุคดั้งเดิมมาใช้อย่างไร

- Live-wire Algorithm
- Saliency Detection
- **Seam Carving** (ใช้ไดนามิกโปรแกรมมิงสองตัวคู่กัน ผลลัพธ์สวยมาก)
- Stereo Algorithm for Solving Correspondence Problem
- Beat Tracking

แรงบันดาลใจ

- Saliency Detection เป็นการเลือกตัดพื้นที่ย่อยที่ติดกัน
- ปัญหามีอยู่ว่า ถ้าวัตถุที่เราสนใจมันอยู่ห่างกันควรจะทำอย่างไรดี
  - อาจจะใช้การตัดพื้นที่ + การย่อรูปเพื่อให้วัตถุที่สนใจทั้งหมดมาอยู่ในพื้นที่
  - เลือกลบแถวหรือคอลัมน์ตรงภาพที่เราไม่สนใจ
    → วัตถุจะย่นระยะเข้ามาหากัน
- แนวคิดแบบที่สองนับว่าน่าสนใจแต่ก็มีปัญหาอยู่ไม่น้อย เพราะภาพจะดูไม่ต่อเนื่อง หรือวัตถุในภาพจะดูผิดรูปไปมาก
- Avidan and Shamir 2007: Content Aware Image Resizing

ตัวอย่างปัญหา



ทำไงดี ถ้าอยากจะเก็บทั้งป้าย โมเดลและพวกหรีด ไว้ด้วยกันในภาพโดยใช้พื้นที่แนวนอนไม่มาก

การเลือกลบเส้นในแนวตั้งออกไป มีโอกาสทำให้วัตถุเสียสัดส่วนไปมาก

Image Credit: Avidan and Shamir, http://www.faculty.idc.ac.il/arik/SCWeb/imret/index.html

แบบนี้มันต้องหาทางล็อคหลบวัตถุกันหน่อยแล้ว

- วัตถุที่เป็นจุดเด่นในภาพมักจะมีขอบที่ชัดเจน
  - กำหนดให้การลบขอบที่ชัดเจนต้องใช้พลังงานที่สูงผิดปรกติ
  - รอยเส้นที่เราทำการลบสามารถขยับซ้ายขวาได้ แต่ต้องลากจากบนลงล่าง และห้ามขาดออกจากกัน → ทำให้ภาพดูต่อเนื่อง

สมมติว่าเราได้ค่าพลังงานมาจากการประมวลผลภาพ

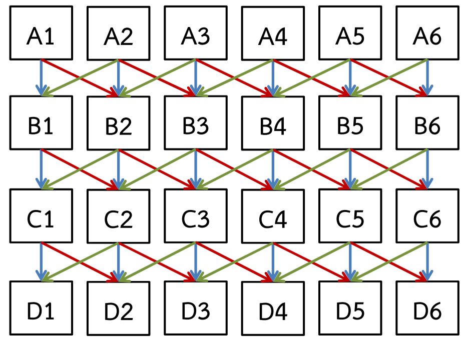
กำหนดพลังงานที่ต้องใช้ในการลบพิกเซลแต่ละอันเป็นไปตามภาพด้านล่าง

จงเสนออัลกอริทึมในการลากเส้นจากขอบบนไปยังขอบล่างที่มี cost น้อยที่สุด โดยที่ในแต่ละแถวจะมีการลบพิกเซลออกหนึ่งพิกเซล นอกจากนี้พิกเซลที่ถูกลบในแถวที่ติดกันจะต้องอยู่ในคอลัมน์เดียวกันหรือติดกันเท่านั้น

7	8	5	1
2	3	5	2
3	3	2	9
9	7	8	9
4	3	5	9
8	6	5	2

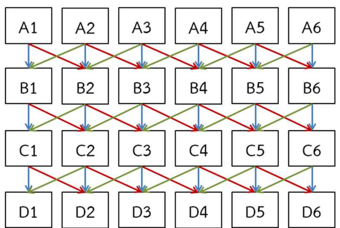
อัลกอริทึมที่พอจะใช้ได้

- เราสามารถใช้ Dijkstra's Algorithm มาช่วยแก้ปัญหาได้ โดยการมองว่าพิกเซลที่ติดกันในแนวที่จะลบออก มีเส้นเชื่อมแบบมีทิศทาง (directed edge) เชื่อมกันอยู่



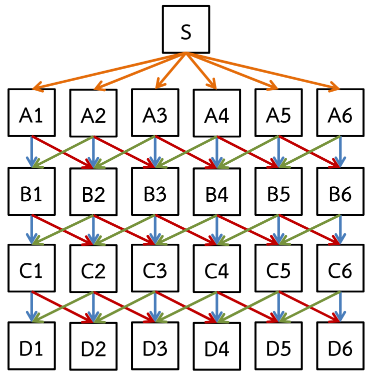
ประสิทธิภาพการคำนวณ

- วิธีคำนวณแบบตรง ๆ บนภาพขนาด M แถว N คอลัมน์จะเริ่มจาก
  - เลือกจุดเริ่มต้นเป็น A1 จากนั้นใช้ Dijkstra's Algorithm
  - คำตอบจะได้มาเมื่อไปถึงแถวสุดท้ายเป็นครั้งแรก จำค่าพลังงานที่ใช้ไว้
  - เปลี่ยนจุดเริ่มเป็น A2, A3, ..., AN แล้วทำแบบเดียวกัน
  - เลือกจุดเริ่มต้นที่ใช้พลังงานน้อยที่สุดเป็นคำตอบสุดท้าย
- เวลาที่ใช้ในการทำงานเป็น $O(MN^2 \log(MN))$
  - ใช้เวลาค่อนข้างมากก็เพราะว่าต้องเรียก Dijkstra's Algorithm มากถึง N รอบ
  - โหนดหลายโหนดถูกนำมาคิดซ้ำซ้อน
  - แต่ที่จริงถ้า N > M ความซ้ำซ้อนจะน้อยกว่ากรณี M > N (ทราบหรือไม่ว่าทำไม ดูภาพข้าง ๆ ประกอบก็พอจะเดาได้)



ลดความซ้ำซ้อน: ใช้ Dijkstra's Algorithm เพียงรอบเดียว

- รวมกราฟให้มีจุดเริ่มต้นเพียงจุดเดียว



- โหนดแต่ละโหนดจะถูกนำมาคิดเพียงครั้งเดียว และจะสรุปเส้นทางที่สั้นที่สุดจากโหนด A ทุกโหนดที่เป็นไปได้ด้วยการใช้ Dijkstra's Algorithm จากโหนด S เพียงรอบเดียว
- เวลาในการคำนวณลดลงมาเหลือเพียง $O(MN \log(MN))$

แต่เรามีแบบที่เร็วกว่านั้น

- จากตัวอย่างจากแผนภาพพลังงานของแต่ละพิกเซล

7	8	5	1
2	3	5	2
3	3	2	9
9	7	8	9
4	3	5	9
8	6	5	2

- สมมติว่าภาพเราเหลือแค่แถวแรก คือให้แถวแรกของภาพเป็นจุดสิ้นสุดด้วย
  → พลังงานที่ใช้ที่น้อยที่สุดของแต่ละพิกเซลก็คือพลังงานทั้งหมดที่จะใช้
- ถ้าเราเพิ่มขอบเขตภาพให้เป็นสองแถวล่ะ

หลักการคิดวิธีที่เร็วขึ้น

- คิดเหมือนค่อย ๆ ขยายขอบเขตภาพขึ้นเรื่อย ๆ

7	8	5	1
2	3	5	2

ผังพลังงาน

7	8	5	1

พลังงานน้อยที่สุด
จากแถวแรก

7	8	5	1
9	8	6	3

พลังงานน้อยที่สุด
จากสองแถวแรก

- ตอนที่เรามีแค่แถวเดียวพลังงานที่ใช้ก็มาจากการเอาพิกเซลนั้น ๆ ออกไป
  - เนื่องจากไม่มีทางเลือกอื่นใดสำหรับพิกเซลนั้น ๆ เราจึงมั่นใจได้ว่าค่าพลังงานที่ใช้ไปนั้นคือค่าที่น้อยที่สุดแล้ว ไม่เป็นอย่างอื่นแน่นอน
- พอมาถึงแถวที่สองเรารู้ว่าเราต้องใช้พลังงานจากการลบแต่ละพิกเซลเช่นเดิม
  - แต่พลังงานรวมที่จะใช้ลบทั้งพิกเซลในแถวแรกและแถวสองล่ะ ที่น้อยที่สุดเป็นเท่าใด
  - เราดูที่แถวแรกแล้วเลือกช่องที่ใช้พลังงานน้อยที่สุดที่ติดกันออกมา

อธิบายเพิ่มเติม

7	8	5	1
2	3	5	2

ผังพลังงาน

7	8	5	1

พลังงานน้อยที่สุด
จากแถวแรก

7	8	5	1
9	8	6	3

พลังงานรวมน้อยที่สุด
จากสองแถวแรก

ตัวเลขแสดงพลังงานรวมของแถวที่สองนั้นคิดมาได้ดังนี้

- คอลัมน์แรก ตัวพิกเซลตำแหน่ง (2, 1) ต้องการพลังงานสองหน่วยในการลบ เนื่องจากต้องลบพิกเซลในแถวแรก จึงต้องเลือกลบพิกเซล (1, 1) หรือ (1, 2) ออกไป
- ในเมื่อมีทางเลือกสองทาง เราก็เลือกทางที่ใช้พลังงานน้อยที่สุดซึ่งก็คือการลบพิกเซล (1, 1) คู่กับการลบพิกเซล (2, 1) จึงใช้พลังงานรวม 7 + 2 = 9 หน่วย
- คอลัมน์สอง ตัวพิกเซลตำแหน่ง (2, 2) ต้องการพลังงานสามหน่วยในการลบ ทำนองเดียวกัน เราต้องเลือกลบพิกเซล (1, 1), (1, 2) หรือ (1, 3) ออกไปด้วย
- ทางที่ใช้พลังงานน้อยสุดคือการลบพิกเซล (1, 3) และพลังงานรวมที่ใช้คือ 5 + 3 = 8 หน่วย

คิดเพิ่มต่อมาเป็นสามแถว

พิจารณาการหาค่าพลังงานที่น้อยที่สุดของพิกเซลแถวที่สาม

- เราสามารถคิดแบบเดียวกันกับที่เราทำในแถวที่สองได้
- เรามั่นใจว่าทำแบบเดิมแล้วจะได้ผลลัพธ์ที่ถูกต้องก็เพราะว่า การจะลบพิกเซลแถวที่สามได้ก็ต้องลบพิกเซลแถวที่หนึ่งและสองออกก่อน
- เพื่อที่จะใช้พลังงานน้อยที่สุด ทางที่ผ่านมาก็ต้องใช้พลังงานน้อยที่สุดด้วย
- ในเมื่อสิ่งที่คิดไว้ในแถวที่หนึ่งใช้พลังงานรวมน้อยที่สุด แถวที่สองก็ใช้พลังงานรวมน้อยที่สุด เราจึงรับประกันได้ว่าแถวที่สามก็ใช้พลังงานรวมน้อยที่สุดเช่นกัน

7	8	5	1
2	3	5	2
3	3	2	9

ผังพลังงาน

7	8	5	1
9	8	6	3

พลังงานน้อยที่สุด
จากสองแถวแรก

7	8	5	1
9	8	6	3
11	9	5	11

พลังงานน้อยที่สุด
จากสามแถวแรก

ไดนามิกโปรแกรมมิงกับการคำนวณพลังงานในการลบพิกเซล

หลักการคิดแบบขยายผลไปเรื่อย ๆ นี้นำไปสู่สมการไดนามิกโปรแกรมมิง

- พิจารณาการหาค่าพลังงานที่น้อยที่สุดของพิกเซลแถว *i* คอลัมน์ *j*
- เราใช้ค่าพลังงานรวมน้อยที่สุดในแถวก่อนหน้ามาบวกกับพลังงานที่ใช้ในการลบพิกเซลตำแหน่ง (i, j) ออกไป
- เนื่องจากพิกเซลที่จะลบต้องติดกัน ซึ่งมีได้แค่สามแบบ สมการที่ได้จึงออกมาเป็น

$$M(i,j) = e(i,j) + \min(M(i-1,j-1), M(i-1,j), M(i-1,j+1))$$

เมื่อ M(i, j) คือ Minimum Energy Sum จากแถวแรกมาจนถึงการลบพิกเซล (i, j)

e(i, j) คือ Energy ที่ต้องใช้ในการลบพิกเซล (i, j) ออกไป

การคำนวณตำแหน่งพิกเซลที่จะลบ

จากวิธีการไดนามิกโปรแกรมมิง เราจะได้พลังงานรวมที่น้อยที่สุดในการลบพิกเซลแต่ละพิกเซลในแถวสุดท้าย

- แต่เราต้องการพลังงานรวมที่น้อยที่สุด เราจึงต้องตามหาพิกเซลในแถวสุดท้ายที่ใช้พลังงานรวมน้อยที่สุด
- ตำแหน่งพิกเซลในแถวสุดท้ายที่ใช้พลังงานรวมน้อยที่สุดนั่นแหละคือตำแหน่งที่ต้องลบ → คำถามจึงมีต่อมาว่าแล้วตำแหน่งพิกเซลในแถวอื่น ๆ ล่ะ
- เราต้องคิดย้อนกลับไปจากตำแหน่งสุดท้าย ซึ่งทำได้โดยการจำไว้ด้วยว่าเราเลือกพิกเซลก่อนหน้ามาจากพิกเซลใด (สมมติว่าภาพมีสองแถวก่อน)

7	8	5	1
2	3	5	2

ผังพลังงาน

7	8	5	1
9	8	6	3

เวลาคำนวณพลังงานรวมน้อยที่สุดของแถวสองให้บันทึกไว้ด้วยว่าเราเลือกพิกเซลแถวก่อนหน้าพิกเซลใด

ลองทำแบบทั้งภาพ

- เนื่องจากตอนแรกเราไม่อาจทราบได้ว่าสุดท้ายใครจะใช้พลังงานรวมน้อยที่สุด → ต้องจำทุกอย่างไว้ก่อนแล้วค่อยเลือกเฉพาะพิกเซลที่ดีที่สุดในตอนหลัง

ผังพลังงาน

7	8	5	1
2	3	5	2
3	3	2	9
9	7	8	9
4	3	5	9
8	6	5	2

พลังงานรวมน้อยที่สุดและเส้นทางทั้งหมด

7	8	5	1
9	8	6	3
11	9	5	12
18	12	13	14
16	15	17	22
23	21	20	19

พลังงานรวมน้อยที่สุดและเส้นทางที่ดีที่สุด

7	8	5	1
9	8	6	3
11	9	5	12
18	12	13	14
16	15	17	22
23	21	20	**19**

การลดขนาดภาพทั้งในแนวตั้งและแนวนอน

- การลดขนาดภาพในแนวในแนวหนึ่งแต่เพียงแนวเดียวเป็นสิ่งที่คำนวณได้โดยง่ายเพราะเป็นการหารอยต่อที่จะลบที่พลังงานน้อยที่สุดไปเรื่อย ๆ
- แต่เมื่อเราต้องการลดขนาดรูปทั้งในแนวตั้งและแนวนอน
  - จะลบแนวไหนก่อนดีหรือว่าจะลบแบบสลับแนว
  - แล้วจะใช้หลักการใดมาเป็นตัวตัดสินทิศทางที่จะลบ?
  - วิธีตัดสินควรจะทำให้เกิดการใช้พลังงานในการลบพิกเซลลดลง ในขณะเดียวกันวิธีคิดก็ควรจะมีประสิทธิภาพพอสมควร
- สมมติว่าเราต้องการลดภาพขนาด M x N (M แถว N คอลัมน์) ให้เป็นภาพขนาด M’ x N’ เราต้องหาลำดับทิศทางในการลบที่ใช้พลังงานน้อยที่สุด
  - เพื่อความสะดวกในการเขียนขอกำหนดให้ M – M’ = R และ N – N’ = C

วิเคราะห์ปัญหา

กำหนดให้มีการลบ R แถว และ C คอลัมน์

- ในลักษณะนี้ ถ้า R = 3, C = 2 เราสามารถทำการเรียงสับเปลี่ยนได้ 10 แบบ
  r r r c c, r r c r c, r c r r c, c r r r c, r r c c r,
  r c r c r, c r r c r, r c c r r, c r c r r, c c r r r
- แต่ในภาพดิจิทัลโอกาสที่ค่า R และ C จะมีมากถึง 100 เป็นเรื่องปรกติ และจำนวนวิธีที่จะเรียงสับเปลี่ยนได้จะมีสูงมาก ดังนั้นเราต้องหาทางอื่น
  - อาจเป็นการประมาณผลลัพธ์ที่สมเหตุสมผลดี ไม่ต้องให้ผลที่แม่นตรงก็ได้
  - ใช้เวลาในการคำนวณไม่มากนัก

ลองกลับมาพิจารณากรณีพื้นฐานเมื่อ R = 1 และ C = 1

- ทางเลือกที่เป็นไปได้มีแค่สองทางคือ (1) ลบแถวก่อน และ (2) ลบคอลัมน์ก่อน
- เราต้องพิจารณาว่าแบบไหนจะใช้พลังงานน้อยที่สุด

พิจารณาทางเลือกทั้งสองแบบ

กำหนดให้ $E_R(a,b)$ และ $E_C(a,b)$ เป็นพลังงานที่ใช้ในการลบแถวและคอลัมน์เมื่อภาพมีขนาด a แถว b คอลัมน์ ทางทั้งสองแบบจะใช้พลังงานดังนี้

- แบบลบแถวก่อน $E_R(M,N) + E_C(M-1,N)$
- แบบลบคอลัมน์ก่อน $E_C(M,N) + E_R(M,N-1)$
- วิธีที่เราควรเลือกคือ
  $$\mathrm{E}(\mathrm{M}-1,\mathrm{N}-1) = \min(E_R(M,N) + E_C(M-1,N), E_C(M,N) + E_R(M,N-1))$$
  เมื่อ $E(a,b)$ คือพลังงานที่น้อยที่สุดในการลดขนาดรูปให้เหลือ a แถว b คอลัมน์
- พูดง่าย ๆ ก็คือเราพยายามเลือกวิธีที่ใช้พลังงานน้อยที่สุดจากสองทางนี้

ตอนนี้เรารู้แล้วว่าถ้าจะหดภาพให้เหลือขนาด $(M-1,N-1)$ เราจะเลือกวิธีไหน คำถามต่อมามีอยู่ว่าหากเราต้องการหดภาพให้เป็นขนาด $(M-2,N-1)$ ล่ะ

ต่อยอดแนวคิดเดิม

- พอเราต้องการจะหดภาพให้เป็นขนาด $(M-2,N-1)$ เราจะพบว่าทางเลือกที่เป็นไปได้มีสามแบบคือ r c r, c r r และ r r c
- สองแนวทางแรกเราเอาผลที่คิดมาก่อนหน้ามาทำต่อได้ทันที
  - r c r ก็คือการหดภาพให้เหลือ $(M-1,N-1)$ ก่อน
  - c r r ก็คือการหดภาพให้เหลือ $(M-1,N-1)$ ก่อน
  - ดังนั้นจากสองแนวทางแรก เราเพียงนำค่า $E(M-1,N-1)$ ที่หามาได้ก่อนหน้ามาคิดต่อรอบเดียวก็เสร็จแล้ว
- ปัญหาที่น่าสนใจก็คือว่า แล้วเราต้องคำนวณค่าจากวิธี r r c จากจุดเริ่มต้นหรือเปล่า
  - ไม่จำเป็น เพราะก่อนหน้าเราคำนวณ $E_R(M,N)$ มาก่อนแล้ว
  - เรามี $E(M-1,N)$ มาก่อนแล้ว เราคำนวณต่ออีกสอง step ก็พอ (คือ r c)
  - ไม่ต้องเริ่มคำนวณ r r c ใหม่ทั้งหมด → แสดงว่าการเก็บผลลัพธ์ไว้ช่วยท่านได้

แล้วรูปแบบทั่วไปในการคำนวณล่ะ

- เช่นเดิม เราเลือกทางที่ให้พลังงานน้อยที่สุดมาเป็นค่า $E(M-2,N-1)$
- ณ จุดนี้เรารู้ค่า E ทั้งหมดดังนี้ (เราเก็บค่า E ไว้ในตารางเพื่อเอามาใช้ต่อทีหลัง)

E(M, N) = 0 (ยังไม่มีการลบใด ๆ)	E(M, N-1)	?1
E(M – 1, N)	E(M-1, N-1)	?2
E(M – 2, N)	E(M-2, N-1)	?3
?1	?2	?4

- ตัว ? สีแดง ๆ ที่ใส่ไว้ระบุว่าต้องคำนวณอีกกี่ขั้นจึงจะได้ผลลัพธ์เป็นค่าพลังงานน้อยที่สุด ซึ่งเราคำนวณเสริมต่อขึ้นไปเรื่อย ๆ ได้ ไม่ต้องคำนวณใหม่ด้วยสมการ
$$E(a,b) = \min(E(a+1,b) + E_R(a+1,b),\ E(a,b+1) + E_C(a,b+1))$$

อธิบายความหมายของสมการ

$$E(a,b) = \min(E(a+1,b) + E_R(a+1,b),$$
$$E(a,b+1) + E_C(a,b+1))$$

มีความหมายว่าพลังงานน้อยที่สุดที่ต้องใช้ในการหดรูปให้เหลือ $E(a,b)$ ซึ่งหามาได้จากความเป็นไปได้สองอย่างคือ

1. $E(a+1,b) + E_R(a+1,b)$ ซึ่งก็คือพลังงานน้อยที่สุดที่ใช้ในการหดรูปให้เหลือขนาด a+1 แถว b คอลัมน์ (ใหญ่กว่าขนาดสุดท้ายหนึ่งแถว) บวกกับพลังงานน้อยที่สุดที่ต้องใช้ในการเอาแถวออกเพิ่มอีกหนึ่งแถวเพื่อให้ได้ขนาดสุดท้าย
2. $E(a,b+1) + E_C(a,b+1)$ ซึ่งก็คือพลังงานน้อยที่สุดที่ใช้ในการหดรูปให้เหลือขนาด a แถว b+1 คอลัมน์ (ใหญ่กว่าขนาดสุดท้ายหนึ่งคอลัมน์) บวกกับพลังงานน้อยที่สุดที่ต้องใช้ในการเอาคอลัมน์ออกเพิ่มอีกหนึ่งคอลัมน์

จากความเป็นไปได้ทั้งสองแบบ เราเลือกทางที่ใช้พลังงานน้อยกว่าก็จะได้คำตอบออกมา

หมายเหตุ E_R และ E_C หามาได้จากวิธีที่สอนไปต้นเรื่องนะ อย่างงว่ามันมาจากไหน

สรุป

- การจะใช้ไดนามิกโปรแกรมมิงเพื่อหาค่าที่ดีที่สุด จะต้องมีปัญหาซ้อนเหลื่อมก่อน ซึ่งเราควรตรวจดูจากสถานะและเซตของ action ที่เหลือ
- เน้นที่การแทนคำตอบจากทางเลือกที่ ‘หมดหวัง’ ด้วยคำตอบที่ ‘ยังมีหวัง’
- ต้องยืนยันได้ว่าคำตอบที่หมดหวังนั้น มันหมดหวังจริง
  → ยืนยันโครงสร้างย่อยเหมาะที่สุด
- ถ้าสังเกตเห็นคุณสมบัติในการตัดคนที่หมดหวังได้เร็ว โค้ดจะมีประสิทธิภาพมาก
- การค้นคืนคำตอบมักต้องอาศัยโครงสร้างข้อมูลอื่นมาช่วย
  - บางรูปแบบเขียนโค้ดยากมาก
- ถ้าไม่จำเป็นไม่ควรสร้างตารางขนาดใหญ่ เพราะใช้เวลาในการสร้างนานและนำไปสู่ปัญหาการใช้หน่วยความจำเกินได้
- งานประยุกต์ใช้ที่สมจริงมักจะเป็นอัลกอริทึมหลายตัวประกอบกัน