Greedy Algorithms

กันต์ ศรีจันท์ทองศิริ

สถาบันเทคโนโลยีนานาชาติสิรินธร

Greedy Algorithm คืออะไร

- อัลกอริทึมสำหรับปัญหาประเภท **optimization** หลาย ๆ ข้อ มักจะประกอบด้วยหลายขั้นตอน
- แต่ละขั้นตอนจะต้องมีการตัดสินใจบางอย่าง
- **Greedy Algorithm** คือประเภทของอัลกอริทึมที่มีแนวคิดว่าในแต่ละขั้น จะตัดสินใจเลือกสิ่งที่**ดูเหมือนเป็นสิ่งที่ดีที่สุด**ในตอนนั้น ๆ เสมอ

- (Greedy = โลภ)

- สำหรับบางปัญหา, Greedy Algorithms จะให้คำตอบสุดท้ายที่ดีที่สุดจริง ๆ (Globally optimal solution).
- แต่หลาย ๆ ปัญหา การใช้ Greedy Algorithms จะไม่ได้คำตอบที่ดีที่สุด
  - แต่ก็อาจจะเป็นคำตอบที่ดีพอสำหรับบางกรณี
    - อย่างเช่นปัญหาเป็น NP-Hard แต่มีวิธี Greedy ง่าย ๆ ที่ให้คำตอบที่ใกล้เคียงกับ Optimal solution

- **ข้อดี:**
  - มักจะง่ายในการเขียนโปรแกรม
  - โปรแกรมมักจะมี Time Complexity ต่ำ
    - เพราะสิ่งที่เลือกไปแล้ว มักจะไม่กลับมาเปลี่ยนการตัดสินใจทีหลัง และ
    - การเลือกสิ่งที่ดีที่สุดในขณะหนึ่ง ๆ มักทำได้ง่าย
- **ข้อเสีย:**
  - มักไม่ได้ global optimal solution ในหลาย ๆ ปัญหา

- เพราะฉะนั้น เมื่อเจอปัญหาใหม่ ๆ ขั้นแรกควรลองดูว่าสามารถใช้หลักการ Greedy ในการแก้ปัญหาได้เลยหรือไม่ และได้คำตอบที่ Optimal หรือไม่ (อาจจะต้องพิสูจน์)
  - ถ้าได้ ก็ใช้เลย
  - ถ้าไม่ได้ ค่อยลองอัลกอริทึมประเภทที่ซับซ้อนกว่าแทน

ปัญหาแคชเชียร์ทอนเงิน

- แคชเชียร์มีเหรียญ 1 บาท, 2 บาท, 5 บาท, และ 10 บาท จำนวนไม่จำกัด (ไม่มีธนบัตร)
- ต้องการทอนเงิน W บาท ให้ใช้จำนวนเหรียญในการทอนน้อยที่สุดต้องทอนอย่างไร

วิธีที่คนทั่วไปใช้:

1. ทอนเหรียญ 10 บาทให้มากที่สุด โดยไม่เกินจำนวนเงินที่ต้องทอน
2. หลังจากนั้น ทอนจำนวนเงินที่เหลือด้วยเหรียญ 5 บาทให้มากที่สุด โดยไม่เกินจำนวนเงินที่ต้องทอน
3. ทำเหมือนเดิมกับเหรียญ 2 บาท และ 1 บาท...

ตัวอย่างการทำงานของวิธีนี้

- ถ้าต้องทอน W = 18 บาท
- ทอนเหรียญ 10 บาทหนึ่งเหรียญ (ถ้า 2 เหรียญจะเกิน 18 บาท)
- เหลือ 8 บาท ทอนเหรียญ 5 บาทหนึ่งเหรียญ (ถ้า 2 เหรียญจะเกิน 8 บาท)
- เหลือ 3 บาท ทอนเหรียญ 2 บาทหนึ่งเหรียญ
- เหลือ 1 บาท ทอนเหรียญบาท หนึ่งเหรียญ
- ทั้งหมดใช้ 4 เหรียญ

- วิธีนี้เป็นวิธี Greedy
  - การทอนโดยเหรียญ 10 บาทให้มากที่สุดโดยไม่เกินจำนวนที่ต้องทอน เป็นสิ่งที่ดูเหมือนดีที่สุดในขณะนั้น (โดยไม่ได้คำนึงถึงเงินที่เหลือที่ต้องทอนด้วยเหรียญอื่น ๆ)

Optimal หรือไม่?

- วิธีนี้ได้คำตอบ Optimal ถ้าเป็นเงินไทย (และเงินสกุลต่าง ๆ ส่วนมากในโลกนี้)
- ยัง Optimal ถึงแม้รวมธนบัตรทั้งหมดเข้าไปด้วย
- เมื่อไหร่ วิธีนี้ถึงจะไม่ Optimal?
  - สมมติว่ามีเหรียญ 4 บาทด้วย

- ถ้าต้องทอน 8 บาท (มีเหรียญ 5 บาท, 4 บาท, 2 บาท, 1 บาท)
- วิธี **Greedy** จะทอนอย่างไร? ใช้กี่เหรียญ?
- คำตอบ **Optimal** คือ?

- สาเหตุที่ไม่มีเหรียญ 4 บาท

Running time

- Running time ของวิธีทอนเงินแบบ greedy คือ O(n)
  เมื่อ n = จำนวนประเภทเหรียญที่มี (ถ้าไม่ต้องเรียงมูลค่าเหรียญ)

พิสูจน์อย่างไรว่า Greedy ได้คำตอบที่ optimal

- โดยทั่วไป ปัญหาหนึ่ง อาจจะมี optimal solution ได้มากกว่าหนึ่ง
- พิสูจน์ว่า มีหนึ่งใน optimal solution ที่เลือกแบบ greedy

พิสูจน์ว่าวิธี Greedy ได้คำตอบ Optimal สำหรับเงินไทย

- จะสมมติว่ามีแค่เหรียญ 1 บาท 2 บาท 5 บาท 10 บาท (ไม่รวมธนบัตร)
  - ถึงรวมธนบัตรก็พิสูจน์คล้าย ๆ กัน

- ข้อสังเกตของ **optimal solution** ใด ๆ ก็ตาม
  - จะใช้เหรียญบาทไม่เกิน 1 เหรียญเสมอ
    - เพราะถ้าใช้ 2 เหรียญขึ้นไป เปลี่ยนเหรียญบาททุก 2 เหรียญเป็นเหรียญบาท 1 เหรียญแทนจะใช้จำนวนเหรียญน้อยกว่า
  - จะใช้เหรียญ 2 บาท ไม่เกิน 2 เหรียญเสมอ
    - ถ้าใช้ 3 เหรียญขึ้นไป เปลี่ยนเหรียญ 2 บาททุก 3 เหรียญ (= 6 บาท) เป็นเหรียญ 5 บาท หนึ่งเหรียญกับเหรียญบาทหนึ่งเหรียญจะดีกว่า
  - จะใช้เหรียญ 5 บาทไม่เกิน 1 เหรียญเสมอ
    - ถ้าใช้ 2 เหรียญขึ้นไป ใช้เหรียญ 10 บาทแทนเหรียญห้าบาททุก 2 เหรียญจะดีกว่า

  - เหรียญบาทไม่เกิน 1 เหรียญ เหรียญ 2 บาทไม่เกิน 2 เหรียญ เหรียญ 5 บาทไม่เกิน 1 เหรียญ
  - จำนวนเงินจากเหรียญ 1, 2, 5 บาท รวมกันไม่เกิน 10 บาท
- เพราะฉะนั้น ถ้าต้องทอนเงินเกิน 10 บาท ต้องใช้เหรียญ 10 บาท จนจำนวนเงินที่เหลือต้องทอน ≤ 10 บาท
  - ถ้าเหลือ 10 บาทพอดี ทอนด้วยเหรียญ 10 บาทอีกเหรียญ จะดีที่สุด (เพราะไม่มีทางทำได้ดีกว่าใช้ 1 เหรียญ)
  - ซึ่งตรงกับวิธี Greedy

- ถ้าเหลือเกิน 5 บาท
  - เหรียญบาท 1 เหรียญ + เหรียญ 2 บาท 2 เหรียญ = 5 บาท
  - ต้องใช้เหรียญ 5 บาทในการทอน
  - ตรงกับ Greedy
- พิสูจน์เหมือนเดิมกับจำนวนที่เหลือ

เหรียญ 4 บาทกับบทพิสูจน์

- มีเหรียญ 4 บาทได้ไม่เกินกี่เหรียญ?
  - **3** เหรียญ = 12 บาท ใช้เหรียญ 10 หนึ่งเหรียญบวกเหรียญสองบาทหนึ่งเหรียญดีกว่า
  - ดังนั้น ไม่เกิน 2 เหรียญ
- เหรียญ 2 บาทกลายเป็นไม่เกิน 1 เหรียญ
- เหรียญบาทไม่เกิน 1 เหรียญ เหรียญ 5 บาทไม่เกิน 1 เหรียญเหมือนเดิม
- จำนวนเงินจากเหรียญ 1, 2, 4 บาท = **11** บาท
- สรุปไม่ได้แล้วว่าถ้าต้องทอนเงินเกิน 5 บาทแต่ไม่ถึง 10 บาท ต้องใช้เหรียญ 5 บาทก่อนเสมอ

Activity Selection

- มีกิจกรรมหลายอย่างที่ต้องการใช้ห้องเดียวกัน
- กิจกรรมอย่างที่ i เริ่มต้นที่เวลา s_i และเสร็จที่เวลา f_i
  - คือใช้เวลาในช่วง $[s_i, f_i)$
- กิจกรรมสองอย่างที่ช่วงเวลาเหลื่อมล้ำกัน จะจัดทั้งสองอย่างไม่ได้
  - ถ้า$[s_i, f_i) \cap [s_j, f_j) \neq \phi$ จะไม่สามารถจัดทั้งกิจกรรม i และ j ได้
- ต้องการเลือกว่าจะจัดกิจกรรมไหนบ้าง เพื่อให้ได้จำนวนกิจกรรมที่จัดได้สูงที่สุด
- เลือกอย่างไร?

- เห็นปัญหาแล้ว อาจจะนึกถึง dynamic programming
  - ทำได้ ได้คำตอบที่ optimal
- แต่มีวิธี greedy ที่ได้คำตอบที่ optimal เหมือนกัน และเร็วกว่า (เทียบ time complexity) (และเขียนโปรแกรมได้ง่ายกว่า)
- Greedy: เลือกแบบไหน?

Greedy

- เลือกจัดกิจกรรมที่เริ่มเร็วที่สุดก่อน (s_i มีค่าน้อยสุด)
- ในขั้นต่อไป เลือกกิจกรรมที่เริ่มเร็วที่สุดจากกิจกรรมที่เหลือ ที่ไม่ชนกับกิจกรรมที่เลือกไปแล้ว

ตัวอย่างการทำงานของวิธีนี้

- มี 3 กิจกรรม: [2, 3), [4, 5), [0, 5)
  - เลือกกิจกรรมที่เริ่มเร็วสุด ก็คือ เลือก [0, 5)
  - หลังจากนั้น เลือกที่เหลือไม่ได้แล้ว
  - ได้จัด 1 กิจกรรม
- แต่ถ้าเลือก [2, 3) กับ [4, 5) จะได้ 2 กิจกรรม

- วิธีนี้ไม่ได้ optimal

Greedy 2

- เลือกกิจกรรมที่ใช้เวลาน้อยที่สุด ($f_i - s_i$ น้อยที่สุด) ที่ไม่ชนกับกิจกรรมที่เลือกไว้แล้ว

- Optimal ไหม?

ตัวอย่างการทำงานของวิธีนี้

- มี 3 กิจกรรม: [0, 100), [99, 102), [101, 1000)
  - เลือกกิจกรรมที่ใช้เวลาน้อยที่สุด ก็คือ เลือก [99, 102)
  - หลังจากนั้น เลือกที่เหลือไม่ได้แล้ว
  - ได้จัด 1 กิจกรรม
- แต่ถ้าเลือก [0, 100) กับ [101, 1000) จะได้ 2 กิจกรรม

- วิธีนี้ไม่ได้ optimal

Greedy 3

- เลือกกิจกรรมที่เสร็จเร็วที่สุด (s_i น้อยที่สุด) ที่ไม่ชนกับกิจกรรมที่เลือกไว้แล้ว

- Optimal ไหม?

- วิธีที่ 3 ได้คำตอบ Optimal จริง

Greedy 3 แบบละเอียด

- เลือกจัดกิจกรรมที่มีเวลาสิ้นสุด f_i เร็วที่สุดในบรรดากิจกรรมที่เหลือ ที่เวลาไม่ชนกับกิจกรรมที่เลือกไว้ครั้งล่าสุด
  - สังเกตว่า ไม่จำเป็นต้องตรวจสอบว่ากิจกรรมใหม่นี้ชนกับกิจกรรมที่เลือกไว้ครั้งก่อนหน้าหรือไม่
- ถ้ามีกิจกรรมที่เข้าข่ายนี้มากกว่าหนึ่ง (เสร็จเวลาเดียวกัน เป็นเวลาเร็วที่สุดในบรรดากิจกรรมที่เหลือ และไม่ชนกับกิจกรรมที่เลือกไว้ล่าสุด) เลือกอันไหนก็ได้

ตัวอย่างการทำงานของวิธีนี้

- มี 5 กิจกรรม: [0, 6), [2, 3), [2, 4), [3, 5), [4, 5)
- ขั้นแรก เลือก [2, 3) เพราะเสร็จเร็วสุด
- ต่อมา งานที่เหลือที่เสร็จเร็วสุดคือ [2, 4)
  - เลือกไม่ได้เพราะชนกับ [2, 3) ที่เลือกไว้แล้วครั้งล่าสุด
- งานที่เวลาเสร็จถัดมาคือ [3, 5) กับ [4, 5)
  - ทั้งสองงานนี้ ไม่ชนกับงานที่เลือกไว้ครั้งล่าสุด ([2, 3))
  - เลือกอันไหนก็ได้ สมมติว่าเลือก [3, 5)

- ต่อมา เหลือ [0, 6) ที่ยังไม่ได้ตัดสินใจ
  - เลือกไม่ได้ เพราะชนกับ [3, 5) ที่เลือกไว้ครั้งล่าสุด
- หลังจากดูครบทั้งหมด คำตอบคือจัดสองงาน: [2, 3) กับ [3, 5)

เพิ่มเติม

- ควรเรียงข้อมูลกิจกรรมตาม f_i จากน้อยไปมากก่อนเริ่ม
- Running time?
  - เรียงข้อมูล = $O(n \log n)$, n = จำนวนกิจกรรมที่มีให้เลือก
  - หลังจากนั้น $O(n)$
    - เพราะแต่ละกิจกรรมตรวจแค่ว่าชนกับกิจกรรมล่าสุดที่เลือกไว้หรือเปล่าเท่านั้น
  - ทั้งหมดคือ $O(n \log n)$

พิสูจน์ว่าวิธี Greedy 3 ได้คำตอบ optimal

- จะพิสูจน์แบบ Induction
- มีกิจกรรม n กิจกรรม ($\{1, 2, \ldots, n\} = S$)
- ให้ $O \subseteq \{1, 2, \ldots, n\}$ เป็น optimal set ของกิจกรรมที่เลือกจัด
  - ตัวอย่าง: $O = \{2, 4, 5\}$ หมายถึงเลือกจัดกิจกรรมที่ 2, 4, และ 5

- สมมติว่ากิจกรรม 1, …, n นี่เรียงตามเวลาเสร็จจากน้อยไปมากแล้ว
  - $f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_n$
  - สามารถสมมติแบบนี้ได้ เพราะถ้า input ไม่เรียง ก็นำมาเรียงแล้วเปลี่ยนชื่อกิจกรรมตามลำดับ

Case 1: ถ้า $1 \in O$, แสดงว่า $O \setminus \{1\}$ เป็น optimal solution ของset กิจกรรม $S' = \{i : s_i \geq f_1\}$

- เพราะ ถ้า $O \setminus \{1\}$ ไม่ใช่ optimal solution ของ S' แสดงว่ามี O' ที่เป็น optimal solution ของ S' และ $|O'| > |O \setminus \{1\}|$ ซึ่งแปลว่า $O' \cup \{1\}$ จะเป็นคำตอบของ S ที่มีจำนวนกิจกรรมมากกว่า O
- ซึ่งแปลว่า O ไม่ optimal... ขัดแย้งกับข้อสมมติ (contradiction)

- ดังนั้น งานแรกของ O เป็นงานเดียวกับงานที่วิธี Greedy 3 เลือก (งานที่เวลาเสร็จเร็วที่สุด)
- ใช้ induction hypothesis ได้ว่า S’ มี optimal solution ที่ได้จากวิธี Greedy 3

Case 2: ถ้า $1 \notin O$.

- ให้ k เป็นงานที่เวลาเสร็จเร็วที่สุดของ O
- Claim: $P = (O \setminus \{k\}) \cup \{1\}$ เป็น set ของกิจกรรมที่ไม่ชนกัน
  - งาน j ใด ๆ ใน $O \setminus \{k\}$ มี $s_j \geq f_k$
    - เพราะ $f_j \geq f_k$. ดังนั้น ถ้า $s_j < f_k$, งาน j จะชนกับงาน k
  - เพราะ $f_1 \leq f_k \Rightarrow f_1 \leq f_k \leq s_j$ นั่นคืองาน j ใด ๆ จะไม่ชนกับงานที่ 1
  - ดังนั้น Claim ถูกต้องแล้ว

- ดังนั้น P เป็นอีก optimal solution หนึ่งของ S เพราะ |P| = |O| และไม่มีกิจกรรมใด ๆ ใน P ชนกัน
- นอกจากนี้ งานแรกของ P เป็นงานเดียวกับงานที่วิธี Greedy 3 เลือก (งานที่เวลาเสร็จเร็วที่สุด)
- ใช้ induction hypothesis ได้ว่า S’ มี optimal solution ที่ได้จากวิธี Greedy 3 เหมือนเดิม

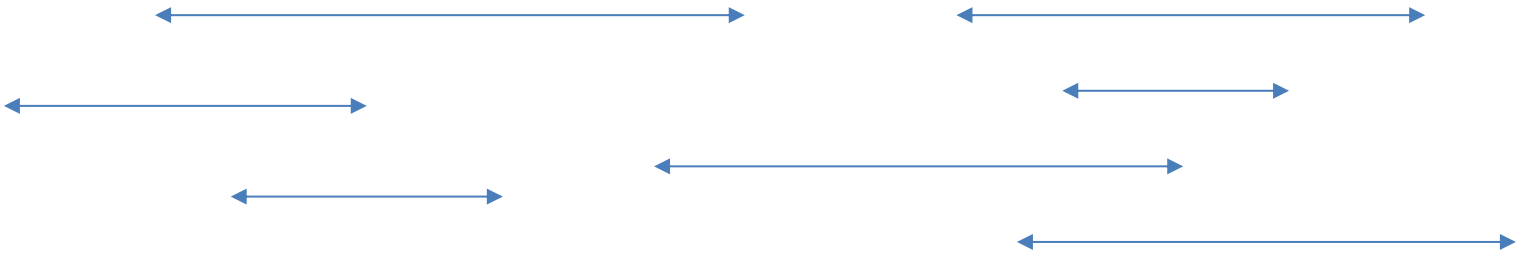
ส่วน Base case ของ induction ก็คือ set {n} กับ ϕ

All Activities Scheduling

- คล้าย Activity Selection
- มีกิจกรรมหลายอย่างที่ต้องการใช้ห้อง
- กิจกรรมอย่างที่ i เริ่มต้นที่เวลา s_i และเสร็จที่เวลา f_i
  - คือใช้เวลาในช่วง $[s_i, f_i)$
- **แต่คราวนี้มีหลายห้อง**

- กิจกรรมที่เวลาเหลื่อมกัน จะจัดในห้องเดียวกันไม่ได้
- คำถาม: **ถ้าต้องการจัดให้ได้ทุกกิจกรรม โดยใช้จำนวนห้องน้อยที่สุด จะจัดอย่างไร**

- **ข้อสังเกต:** ถ้ามีเวลาขณะหนึ่ง ๆ ที่มีกิจกรรม 3 อย่างเกิดขึ้นพร้อมกัน ก็ต้องใช้อย่างน้อย 3 ห้องในการจัดกิจกรรม
- **นิยาม:** ความลึก (depth) ของกลุ่มกิจกรรม คือ จำนวนกิจกรรมที่มากที่สุดที่มีช่วงเวลาคร่อมเวลา ณ จุดหนึ่ง



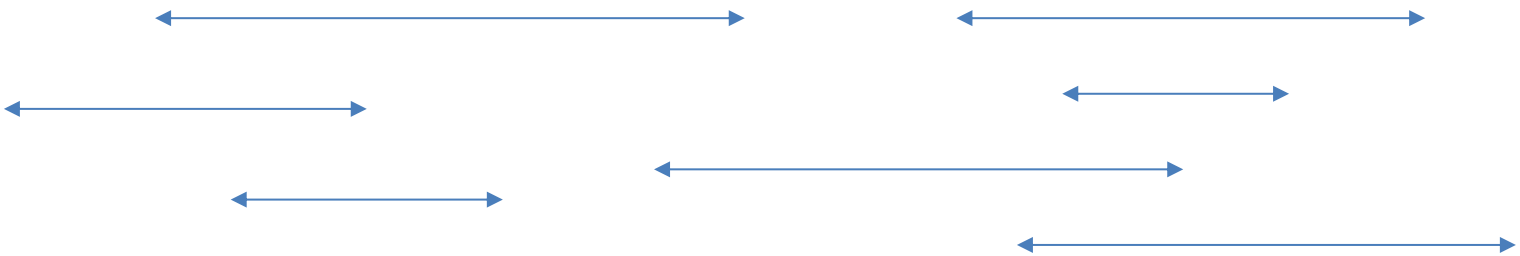
- ความลึก = 4

- **ข้อสังเกต: ไม่สามารถจัดตารางโดยใช้จำนวนห้องน้อยกว่าความลึกของกลุ่มกิจกรรมได้แน่ ๆ**
- => ถ้าสามารถจัดตารางโดยใช้จำนวนห้องเท่ากับความลึกได้ ตารางนั้นจะ optimal แน่นอน

Greedy algorithm สำหรับปัญหานี้

- จะจัดโดยใช้จำนวนห้องเท่ากับความลึกของกลุ่มกิจกรรม (d)
- สมมติว่ามีห้องเบอร์ 1, 2, ..., d เท่านั้น
- เรียงกิจกรรมตามเวลาเริ่ม s_i โดยกิจกรรมที่เวลาเริ่มพร้อมกัน เรียงอันไหนก่อนก็ได้
- จะเลือกห้องให้กิจกรรมตามลำดับนี้
- กิจกรรมแรก (ที่เวลาเริ่มเร็วที่สุด) จัดใส่ห้องไหนก็ได้
- กิจกรรมต่อ ๆ ไป จัดใส่**ห้องไหนก็ได้ที่ว่างในเวลาที่กิจกรรมนี้เริ่ม**

ลองกับปัญหานี้



- ความลึก = 4 เลยใช้ 4 ห้อง

- ต้องพิสูจน์ว่า Algorithm นี้ หาห้องให้ทุกกิจกรรมได้ โดยใช้แค่ d ห้องจริง ๆ
- “กิจกรรมต่อ ๆ ไป จัดใส่**ห้องไหนก็ได้ที่ว่างในเวลาที่กิจกรรมนี้เริ่ม**”
- ต้องพิสูจน์ว่าขั้นตอนข้างบนนี้ จะมีห้องว่างให้จัดกิจกรรมอันที่กำลังตัดสินใจได้เสมอ

- สมมติว่าไม่จริง
- แปลว่า มีกิจกรรมหนึ่ง (สมมติว่ากิจกรรมที่ ***i***) ที่เมื่อถึงเวลาที่เราจะจัดห้องให้ แต่ห้องทั้ง d ห้องเต็มทั้งหมด
- แปลว่ามีกิจกรรม d+1 อย่างที่มีเวลาคร่อมเวลา s_i
- ขัดกับที่เราตั้งไว้ว่า d คือความลึกของกลุ่มกิจกรรม...

ปัญหา Fractional Knapsack

- ขโมยมีถุงใบหนึ่ง ที่รับน้ำหนักได้เต็มที่ W
- มีกองของมีค่าอยู่หลายประเภท
- กองที่ i มีค่า v_i และมีน้ำหนัก w_i
- ต้องการตัดสินใจว่าจะเอาของมีค่าจากกองไหน กองละเท่าไหร่บ้างใส่ถุง โดยที่ถุงไม่ขาด และได้มูลค่ามากที่สุด
- ถ้าเลือกหยิบแค่น้ำหนัก x จากกอง i ให้ถือว่ามีค่า $v_i * x / w_i$
  - อย่างเช่น ของที่มีค่า 4 และหนัก 2 ถ้าเลือกแค่น้ำหนัก 1 จะได้มูลค่า 2

Greedy

- เลือกของที่มี **vi** สูงสุดก่อน ใส่จนเต็มถุงหรือหมดกอง
- หลังจากนั้น ถ้าถุงยังไม่เต็ม เลือกของที่มี **vi** สูงสุดรองลงมา
- ทำแบบนี้จนถุงเต็ม หรือของหมด

- Optimal?

- มีของ 2 กอง ถุงใส่ได้ W = 100
  - $v_1 = 200, w_1 = 1000$
  - $v_2 = 100, w_2 = 100$
- วิธีนี้ เลือกกองที่ 1 ก่อน เพราะมูลค่าสูงกว่า
- ใส่ได้ 1/10 กอง ถุงเต็ม
- ได้มูลค่า = 20
- ไม่ optimal. ถ้าเลือกกองที่ 2 จะใส่ได้ทั้งกอง และได้มูลค่า 100

Greedy 2

- เลือกกองที่มีมูลค่าต่อหน่วยสูงสุด (v_i/w_i) ก่อน
  - ถ้าเสมอ เอาอันไหนก็ได้
- ใส่จนเต็มถุงหรือหมดกอง
- ถ้าถุงยังไม่เต็ม ก็ทำเหมือนเดิมจนถุงเต็ม (หรือของหมด)

- มีของ 2 กอง ถุงใส่ได้ $W = 100$
  - $v_1 = 200, w_1 = 1000$
  - $v_2 = 100, w_2 = 100$
- วิธีนี้ เลือกกองที่ 2 ก่อน เพราะมูลค่าต่อหน่วยสูงกว่า
  - $v_2/w_2 = 1 > 0.2 = v_1/w_1$
- ใส่กองที่ 2 ได้ทั้งหมด ถุงเต็มพอดี
- ได้มูลค่าทั้งสิ้น 100

Running time

- O(n log n) เมื่อ n = จำนวนประเภทกองของที่มี
- เพราะต้องเรียงของตามมูลค่าต่อหน่วย

พิสูจน์ว่า optimal

- สมมติว่ามี optimal solution ที่ไม่เลือกแบบเดียวกับที่ greedy เลือก
- ให้ V คือมูลค่าในถุงของคำตอบ optimal นี้
- ซึ่งก็คือ ถุงเต็มแล้ว และมีของกอง j เหลืออยู่ที่ $v_j/w_j \geq v_i/w_i$ เมื่อ i คือของสักกองหนึ่งที่อยู่ในถุง
  - อาจจะเท่ากันได้ กรณีที่มูลค่าต่อหน่วยเสมอ แต่เลือกกันคนละอย่าง

- เอาของกอง **i** ออกเท่ากับน้ำหนัก $e > 0$ และใส่กอง **j** ด้วยน้ำหนักเดียวกันเข้าไปแทน
  - สังเกตว่า $e \le \min($น้ำหนักของกอง **j** ที่เหลือ, น้ำหนักของกอง **i** ในถุง$)$
- หลังจากนั้น มูลค่าในถุงกลายเป็น

$$V - e(v_i/w_i) + e(v_j/w_j) \ge V$$

เพราะ $v_j/w_j \ge v_i/w_i$.

ปัญหา 0-1 Knapsack

- เปลี่ยนปัญหา **fractional knapsack** เป็นว่า ของเป็นชิ้น ไม่ได้เป็นกอง แบ่งไม่ได้
  - คือ ถ้าเลือกของชิ้นที่ **i** ต้องใส่ถุงทั้งชิ้น ใส่แค่ครึ่งชิ้นไม่ได้
- ทำอย่างไร? **Greedy** ยัง **optimal** อยู่ไหม?

- มีของ 3 ชิ้น ถุงใส่ได้ W = 10
  - $v_1 = 8, w_1 = 6.\ v_1/w_1 = 4/3.$
  - $v_2 = 5, w_2 = 5.\ v_2/w_2 = 1.$
  - $v_3 = 4, w_3 = 5.\ v_3/w_3 = 4/5.$
- Greedy จะเลือกของชิ้นที่ 1 ก่อน
- หลังจากนั้น จะใส่อะไรไม่ได้อีกแล้ว
- มูลค่าทั้งหมด = 8
- แต่ถ้าไม่เลือกชิ้นที่ 1 แต่เลือกชิ้นที่ 2 กับ 3 แทน จะได้มูลค่า 9 ซึ่งมากกว่า

- ต้องทำอย่างไรถึงจะ Optimal?
- Dynamic programming. บ่ายนี้

ระบบปฏิบัติการ: CPU Scheduling

- มี CPU หนึ่งตัว ที่รัน process ได้ทีละ process
- มี process เข้ามาพร้อมกันหลายอันในเวลาที่ 0 แต่ละอันใช้เวลาต่างกัน (CPU burst time)
- **คำถาม:** ต้องจัดลำดับให้ CPU รัน process ไหน ก่อนหลัง ลำดับอย่างไร จึงจะมี average waiting time ต่ำที่สุด
  - Waiting time ของ process หนึ่ง คือเวลาที่ process นั้นต้องรอก่อนที่จะได้เริ่มรันบน CPU

- ไม่รวมเวลาที่ process รันบน CPU
- อย่างเช่น process P1 ได้เริ่มรันตอน 5 ms หลังจากเข้ามารอ และใช้เวลารันจนเสร็จ 2 ms จะนับว่า waiting time คือ 5 ms
- Average waiting time คือ average ของ waiting time ของทุก process

Shortest-Job-First (SJF)

- เป็นอัลกอริทึมสำหรับ CPU Scheduling
- เลือกรัน process ที่ burst time ต่ำที่สุดก่อน
  - ถ้าเสมอ เลือกอันไหนก็ได้
- เป็น Greedy
- Optimal ให้ average waiting time ที่ต่ำที่สุด

ตัวอย่าง

- มี 3 process
  - P1 มี burst time = 5 ms
  - P2 มี burst time = 1 ms
  - P3 มี burst time = 2 ms
- SJF: ทำ P2 -> P3 -> P1
- Waiting time ของ P1 = 3 ms
- Waiting time ของ P2 = 0 ms
- Waiting time ของ P3 = 1 ms
- Average waiting time = 4/3 ms

พิสูจน์ว่า Optimal

- ให้ b_i = burst time ของ process i (Pi)
- สมมติว่ามี optimal schedule O ที่ไม่เป็นแบบ SJF
- แปลว่ามีอย่างน้อยสอง process Pi, Pj ที่ schedule ติดกันที่
  - Pi ได้รันก่อน Pj แต่ $b_i \geq b_j$ และ Pj ได้รันทันทีหลังจาก Pi เสร็จ

- สลับลำดับ Pi กับ Pj ใน schedule O
  - เรียก schedule ใหม่ว่า O’
- ไม่กระทบ waiting time ของ process อื่น (ทำไม?)
  - เพราะหลังจากสลับแล้ว Pi จะเสร็จเวลาเดียวกับที่ Pj เสร็จใน schedule เก่า
  - Process หลังจากนี้ ได้เริ่มทำงานเวลาเดิม
  - Process ก่อนหน้าก็เหมือนเดิม

- Waiting time ของ Pj ใน O’ = Waiting time ของ Pi ใน O
- Waiting time ของ Pi ใน O’ = Waiting time ของ Pj ใน O – (bi – bj) ≤ Waiting time ของ Pj ใน O (เพราะ $b_i \geq b_j$)
- ดังนั้น average waiting time ของ O’ ≤ average waiting time ของ O

- ถ้า O’ ยังไม่เป็น SJF ก็เลือกคู่ Pi, Pj คู่ใหม่ที่เป็นตาม SJF
- สลับ Pi กับ Pj เหมือนเดิม
- ทำแบบนี้ไปเรื่อย ๆ สุดท้ายจะได้ schedule ที่เป็นแบบ SJF และมี average waiting time ที่น้อยกว่าหรือเท่ากับ O
- สรุปได้ว่า SJF Schedule เป็น optimal

หมายเหตุกับ SJF

- ในระบบปฏิบัติการจริง ทำไม่ได้ เพราะไม่รู้ได้แน่นอนว่า burst time ของ process หนึ่ง ๆ จะเป็นเท่าใด ก่อนจะเรียก system call หรือ terminate
  - แต่สามารถประมาณได้ เป็น approximate SJF
  - ถึงจะสมมติว่ารู้ burst time ที่แท้จริงได้ ในระบบจริง ต้องคำนึงถึงปัจจัยอื่นนอกจาก waiting time ด้วย ก็อาจจะเลือกวิธีอื่นแทน

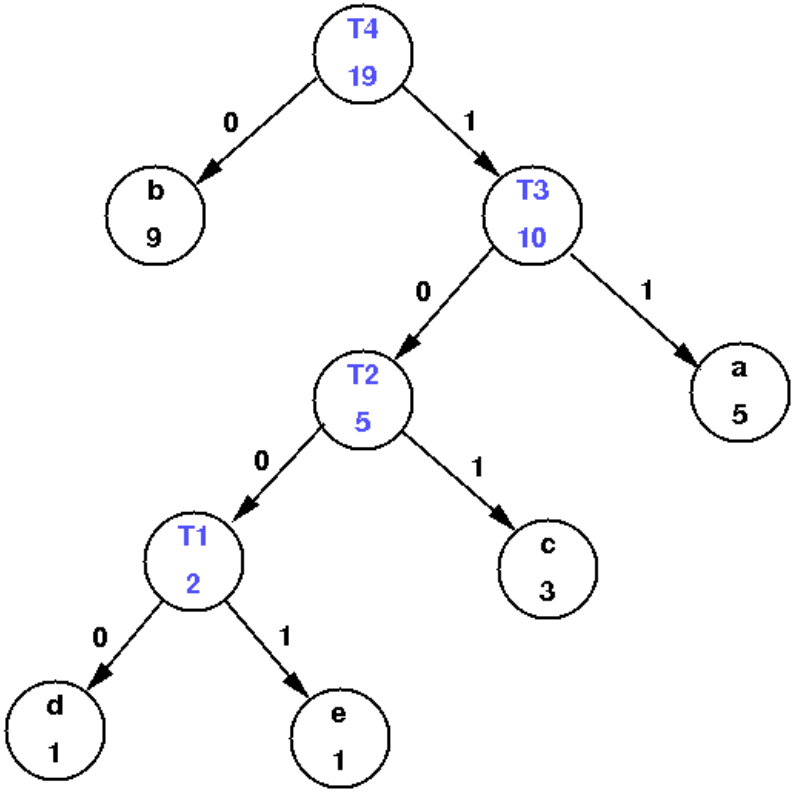
Huffman Coding

- ตามปกติ รหัส Unicode หรือ ASCII ใช้จำนวนบิตเท่ากันสำหรับทุกตัวอักษร
  - e ใช้ 8 bits. z ก็ใช้ 8 bits. (ASCII)
- แต่ e มักปรากฏบ่อยกว่า z
  - Variable-length coding: ถ้าใช้จำนวนบิตแทนตัวอักษรที่พบบ่อย (e) น้อยกว่าจำนวนบิตแทนตัวอักษรที่ไม่ค่อยพบ (z), จะใช้พื้นที่ในการเก็บข้อความทั้งหมดได้น้อยลง
  - (ไม่สามารถลดจำนวนบิตของทุกตัวอักษรได้ บางตัวต้องมี code ที่ยาวขึ้น)

- สมมติว่า e = 10, z = 11 (fixed-length codes)
  eezeee = 101011101010 => 12 บิต
- แต่อาจจะใช้ e = 0, z = 1111 (variable-length codes)
  eezeee = 001111000 => 9 บิต

- นอกจากนี้ การใช้ variable-length coding มีเงื่อนไขว่า code ของตัวอักษรหนึ่ง ๆ ต้องไม่เป็น prefix ของ code ของตัวอักษรอื่น
- a = 1, b = 11
- Code = 111 ...ไม่รู้ว่าเป็น aaa หรือ ab หรือ ba

ใช้ binary tree แทน code ของแต่ละตัวอักษร



รหัสของแต่ละตัวอักษรจาก tree

b = 0

a = 11

c = 101

d = 1000

e = 1001

- การใช้ binary tree โดยให้ตัวอักษรอยู่ที่ leaf เท่านั้น ทำให้มั่นใจได้ว่ารหัสของตัวอักษรหนึ่ง จะไม่เป็น prefix ของรหัสของตัวอักษรอื่น

- **ปัญหา:** มีข้อความยาว ๆ ข้อความหนึ่ง และรู้ว่าตัวอักษรแต่ละตัวปรากฏกี่ครั้งในข้อความนี้ (frequency) ควรจะให้รหัสตัวอักษรต่าง ๆ เป็นอะไร เพื่อที่ใช้รหัสเก็บข้อความนี้แล้ว ใช้พื้นที่น้อยที่สุด
  - หรืออีกอย่างก็คือ สร้าง tree รหัสอย่างไร

Huffman Coding

- ความถี่ของแต่ละตัวอักษร:
- a = 4
- b = 2
- c = 1
- d = 8

- เริ่มให้ทุกตัวอักษรเป็น tree ที่มี node เดียว
- ให้ความถี่ของตัวอักษรเป็นค่าของ node นั้น
- หยิบ 2 tree ที่มีความถี่ต่ำสุดมาเชื่อมกัน โดยสร้าง node ใหม่ขึ้นมา และให้ root ของ 2 tree นี้เป็นลูกทั้งสองของ node ใหม่
- ให้ค่าความถี่ของ node ใหม่เท่ากับผลรวมของความถี่ของลูกทั้งสอง
- ทำไปเรื่อย ๆ จนทั้งหมดรวมเป็น tree เดียว

- สังเกตว่า เป็นวิธี greedy
- นอกจากนี้ ยังได้คำตอบที่ optimal ด้วย

Greedy algorithms ใน Graph Theory

- สัปดาห์หน้า
- Kruskal's กับ Prim's algorithms สำหรับหา minimum spanning tree
- Dijkstra's algorithm สำหรับหา shortest path ใน graph กรณีที่ไม่มี edge ที่มีน้ำหนักเป็นลบหรือศูนย์

แบบฝึกหัด

- http://acm.tju.edu.cn/toj/showp1015.html
- http://acm.tju.edu.cn/toj/showp1805.html
- http://acm.tju.edu.cn/toj/showp1883.html
- http://acm.tju.edu.cn/toj/showp2290.html
- http://acm.tju.edu.cn/toj/showp2708.html

- http://poj.org/problem?id=1548
- http://poj.org/problem?id=1936
- http://poj.org/problem?id=2590
- http://poj.org/problem?id=2620
- UVa: uva.onlinejudge.org/
- UVa Problem Set Volume 1: P311 - Packets

- http://acm.tju.edu.cn/toj/showp2563.html
- UVa Problem Set Volume 1: P120 - Stacks of Flapjacks