

# Greedy Algorithm

# Today's topics

- What is Greedy Algorithm
- Coin Changing Problem
- Fractional Knapsack Problem
- CPU Scheduling Problem
- Activity Selection Problem
- Huffman Coding
- Programming Questions

# What and Why Algorithm?

- Algorithm คือ กระบวนการที่ถูกใช้ในการแก้ปัญหา
- จะเห็นได้ว่าจุดประสงค์หลักของ algorithm คือ การแก้ไขปัญหา
- การเรียนรู้เกี่ยวกับ algorithm จะช่วยให้สามารถเลือกใช้ algorithm ในการแก้ปัญหาได้ดี (ปัญหาส่วนใหญ่สามารถใช้ algorithm หลายประเภทในการแก้ไขปัญหา)
- สิ่งสำคัญที่ต้องจำไว้ในการเลือกใช้ algorithm คือ
  - แก้ไขปัญหาได้ถูกต้อง (give correct output)
  - มีประสิทธิภาพ (efficient)

# Greedy Algorithm

- Greedy แปลว่า โลภ
- Greedy algorithm เป็นกระบวนการในการแก้ปัญหาแบบหนึ่งที่แก้ไข  
ปัญหาแบบทีละขั้นตอน โดยเลือกตัวเลือกที่ดูเหมือนจะดีที่สุด ณ ตอนนั้น  
โดยไม่ได้อคิดทุกอย่างให้รอบคอบ
- ฟังก์ชันดูเหมือนจะไม่ดี แต่ปัญหาหลายข้อสามารถ  
แก้ไขได้โดยหลักการของ greedy algorithm  
และสามารถให้ประสิทธิภาพในการทำงานได้ดีด้วย



# Greedy Algorithm (ต่อ)

- ลักษณะของปัญหาที่เหมาะสมต่อการใช้ Greedy Algorithm
  - Greedy Choice Property: ปัญหาใหญ่สามารถถูกแก้ได้โดยการแก้ไขปัญหาย่อย ๆ ทีละขั้น (ไม่จำเป็นต้องมองแบบองค์รวม)
  - Optimal Sub Structure: คำตอบที่เป็น optimal ของปัญหาเล็ก ๆ สามารถรวมเป็นคำตอบที่ optimal ของปัญหาใหญ่ได้

# ข้อดีของ Greedy Algorithm

- Greedy algorithm นั้นมีคุณสมบัติที่ดีอยู่ด้วยกัน 2 ข้อหลัก
  - Simple: Greedy algorithm เป็น algorithm ที่ง่ายต่อการอธิบายและก็ง่ายแก่การเขียนโปรแกรม
  - Efficient: โดยปกติการใช้ Greedy algorithm จะให้ประสิทธิภาพที่ดีกว่า algorithm ชนิดอื่น

# ความท้าทายของ Greedy Algorithm

- Greedy algorithm นั้นมี 2 ข้อหลัก
  - Design: การออกแบบ Greedy algorithm ให้สามารถทำงานแล้วได้คำตอบที่ optimal นั้นเป็นเรื่องไม่่ง่ายนัก
  - Proof: การพิสูจน์ว่า Greedy algorithm ที่ออกแบบนั้นสามารถทำได้อย่าง optimal ก็ไม่่ง่ายเช่นกัน

การแก้ไขปัญหาด้วย Greedy algorithm



# ปัญหาแลกเหรียญ

- สมมติเราต้องการเขียนโปรแกรมที่ใช้ในการแลกเหรียญ (ชนิดของเหรียญ คือ 1 บาท, 2 บาท, 5 บาท, 10 บาท) โดยให้ใช้จำนวนเหรียญที่น้อยที่สุด
- ตัวอย่าง
  - จ่ายเงิน 15 บาทโปรแกรมจะทอนเงินเป็น
    - 10 บาท 1 เหรียญ
    - 5 บาท 1 เหรียญ
  - จ่ายเงิน 8 บาทโปรแกรมจะทอนเงินเป็น
    - 5 บาท 1 เหรียญ
    - 2 บาท 1 เหรียญ
    - 1 บาท 1 เหรียญ

# การออกแบบ algorithm

- ถ้าทำตามสัญชาตญาณ เราก็จะทอนเงินโดยเลือกเหรียญที่มีมูลค่ามากที่สุดไปก่อน แล้วค่อยทอนด้วยเหรียญที่มีมูลค่าน้อยลงตามมูลค่าที่เหลือ
- ตัวอย่างเช่น หากมีคนต้องการแลกเหรียญ 16 บาท
  - ทอนด้วยเหรียญ 10 ก่อน เหลือมูลค่าที่ต้องทอนอีก 6 บาท
  - ทอนด้วยเหรียญ 5 ต่อมา เหลือมูลค่าที่ต้องทอนอีก 1 บาท
  - ทอนด้วยเหรียญ 1 บาท ครบจำนวนมูลค่าที่ต้องทอน
- นี่เป็นตัวอย่างของ Greedy algorithm เนื่องจากเลือกตัวเลือกที่ดูเหมือนจะดีที่สุดก่อนในการตอบคำถาม

# Greedy algorithm ที่ออกแบบนั้น optimal ไหม?

- คำตอบที่ถูกต้องของ algorithm คือสิ่งสำคัญที่สุด
- คำตอบที่ถูกต้องต้องใช้จำนวนเหรียญที่น้อยที่สุดในการทอน
- จะพิสูจน์ยังไม่ว่า algorithm ที่เราออกแบบนั้นให้คำตอบที่ถูกต้อง (optimal) ?

# พิสูจน์ว่า algorithm นั้น optimal

- ต้องคิดย้อนกลับไปที่คำตอบที่ถูกต้องมีลักษณะอย่างไร
- มีเหรียญอยู่ 4 ชนิด (1 บาท, 2 บาท, 5 บาท, และ 10 บาท)
- ต้องใช้เหรียญ 1 บาท ไม่เกิน 1 เหรียญ (ไม่จำเป็นต้องใช้เหรียญ 2 บาทดีกว่า)
- ต้องใช้เหรียญ 2 บาท ไม่เกิน 2 เหรียญ (ถ้าใช้ 2 บาท 3 เหรียญ = 6 ไปใช้ 5 บาท กับ 1 บาทดีกว่า ใช้แค่ 2 เหรียญ)
- ต้องใช้เหรียญ 5 บาท ไม่เกิน 1 เหรียญ (ถ้าใช้ 5 บาท 2 เหรียญ = 10 ไปใช้เหรียญ 10 บาทดีกว่า)
- ใช้เหรียญ 10 บาทได้ไม่จำกัด แต่ต้องเลือกใช้เหรียญ 10 ก่อนเหรียญอื่น

## เหรียญ 4 บาท

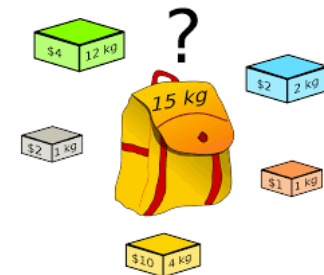
- คำถามคือ ถ้ามีเหรียญ 4 บาทด้วย ปัญหานี้จะสามารถถูกแก้ได้โดย Greedy algorithm แบบเดิมหรือไม่?

# พิสูจน์เหรียญ 4 บาท

- ทอนเงิน 8 บาท
- ตามวิธี Greedy ของเรา ต้องเลือกเหรียญ 5 บาทมาทอนก่อน
- คำตอบที่จะได้คือ (5,2,1)
- แต่เนื่องจากเรามีเหรียญ 4 บาท
- คำตอบที่ควรจะได้คือ (4,4)
- ดังนั้น Greedy algorithm ไม่ได้ให้ผลลัพธ์ที่เป็น optimal เสมอไปถ้ามีเหรียญ 4 บาท
- ต้องใช้ dynamic programming

# ปัญหา Fractional Knapsack

- สมมติเราต้องการเขียนโปรแกรมที่ใช้ในการเลือกหยิบของใส่กระเป๋าให้ได้ค้่มค่าที่สุด โดยมีเงื่อนไขว่า น้ำหนักที่กระเป๋าได้รับได้มีจำกัด และของแต่ละชนิดมีมูลค่าและน้ำหนักที่ไม่เท่ากัน
- สมมติให้
  - กระเป๋าสามารถรับน้ำหนักได้ 300 กิโลกรัม
  - ของกองที่ 1 มีน้ำหนัก (w) 110 กิโลกรัม มีมูลค่า (v) 110 บาท
  - ของกองที่ 2 มีน้ำหนัก (w) 280 กิโลกรัม มีมูลค่า (v) 250 บาท
  - ของกองที่ 3 มีน้ำหนัก (w) 200 กิโลกรัม มีมูลค่า (v) 200 บาท
  - ไม่จำเป็นต้องเลือกของจากกองเดียวกันทั้งหมด เลือกบางส่วนก็ได้



# ใช้ Greedy algorithm ในการแก้ปัญหา

- ครั้งที่ 1 เลือกกองที่มีมูลค่า ( $v$ ) มากที่สุดก่อน
- จะเลือกกองที่ 2 มีมูลค่า 250 และกองที่ 3 (20) จะได้มูลค่ารวม 270
- ใช้คำตอบที่ optimal ไหม?

กระเป๋าสารับน้ำหนักได้ 300 กิโลกรัม

ของกองที่ 1 มีน้ำหนัก ( $w$ ) 110 กิโลกรัม มีมูลค่า ( $v$ ) 110 บาท

ของกองที่ 2 มีน้ำหนัก ( $w$ ) 280 กิโลกรัม มีมูลค่า ( $v$ ) 250 บาท

ของกองที่ 3 มีน้ำหนัก ( $w$ ) 200 กิโลกรัม มีมูลค่า ( $v$ ) 200 บาท



# ใช้ Greedy algorithm ในการแก้ปัญหา

- ครั้งที่ 2 เลือกกองที่มีมูลค่าต่อน้ำหนักสูงสุด ( $v/w$ )
- กองที่ 1 ( $v/w$ ) =  $110/110 = 1$
- กองที่ 2 ( $v/w$ ) =  $250/280 = 0.89$
- กองที่ 3 ( $v/w$ ) =  $200/200 = 1$
- ดังนั้นเลือกกองที่ 1 (100) และกองที่ 3 (200) ได้มูลค่ารวม 300

# พิสูจน์ว่าคำตอบที่ได้ optimal

- เนื่องจาก  $v/w$  = อัตราส่วน มูลค่า/น้ำหนัก
- ข้อจำกัดของถุง คือ น้ำหนักที่ถุงสามารถรับได้  $w$
- สิ่งที่ต้องการคือ มูลค่าของสิ่งของ  $v$
- ให้มอง  $w$  เป็นเหมือน เงินที่จ่ายไป
- ให้มอง  $v$  เป็นเหมือน มูลค่าสิ่งของที่ได้รับมา
- เราใช้วิธีเลือกสิ่งของที่มีมูลค่ามากที่สุด ในขณะที่จ่ายเงินน้อยที่สุด ดังนั้น คำตอบเป็น optimal

# ปัญหา 0-1 Knapsack

- ปัญหาเดิม
  - กระเป๋าสารับน้ำหนักได้ 300 กิโลกรัม
  - ของกองที่ 1 มีน้ำหนัก ( $w$ ) 110 กิโลกรัม มีมูลค่า ( $v$ ) 110 บาท
  - ของกองที่ 2 มีน้ำหนัก ( $w$ ) 280 กิโลกรัม มีมูลค่า ( $v$ ) 250 บาท
  - ของกองที่ 3 มีน้ำหนัก ( $w$ ) 200 กิโลกรัม มีมูลค่า ( $v$ ) 200 บาท
- แต่ครั้งนี้ห้ามแบ่งส่วน นั่นคือ ถ้าจะเลือกหยิบ ต้องหยิบทั้งกอง

# แก้ปัญหาด้วย Greedy algorithm

- โจทย์
  - กระเป๋าสารับน้ำหนักได้ 300 กิโลกรัม
  - ของกองที่ 1 มีน้ำหนัก ( $w$ ) 110 กิโลกรัม มีมูลค่า ( $v$ ) 110 บาท
  - ของกองที่ 2 มีน้ำหนัก ( $w$ ) 280 กิโลกรัม มีมูลค่า ( $v$ ) 250 บาท
  - ของกองที่ 3 มีน้ำหนัก ( $w$ ) 200 กิโลกรัม มีมูลค่า ( $v$ ) 200 บาท
- เลือก  $v/w$  ที่มากที่สุดก่อน
  - 1 และ 3 เท่ากัน สมมติเลือก 3 ก็จะได้มูลค่า 200 บาท (เหลือพื้นที่ 100 กิโลกรัม ใส่อะไรไม่ได้อีกแล้ว)
  - ดังนั้น Greedy algorithm ใช้ไม่ได้กับปัญหานี้

# CPU scheduling problem

- ระบบปฏิบัติการเป็น software ที่ทำหน้าที่ในการติดต่อกับ CPU เพื่อเลือก processes ที่กำลังทำงานอยู่ให้ CPU ประมวลผล
- จะเลือก process ไหนมาทำงานเป็นหน้าที่ของ CPU scheduler
- CPU scheduler ที่ดีจะต้องทำให้ process มีค่าเฉลี่ยของ waiting time ที่น้อยที่สุด
- Waiting time คือเวลาที่ process ต้องรอก่อนที่จะได้รับการประมวลผล
- ตัวอย่างเช่น หาก process 1 พร้อมทำงาน ณ เวลา  $0t$  แต่ CPU scheduler สั่งให้ CPU ทำงาน process 1 ณ เวลา  $5t$  waiting time ของ process 1 ก็จะเป็น  $5t$

# CPU scheduling problem

- สมมติให้ปัจจุบันมีทั้งหมด 3 processes ที่รอทำงานอยู่ในระบบ
- ทั้ง 3 processes มี burst time (เวลาที่ใช้ในการทำงาน) ดังนี้
  - P1: burst time = 5s
  - P2: burst time = 3s
  - P3: burst time = 2s
- จะจัดลำดับการทำงานของ process อย่างไรให้ได้ average waiting time น้อยที่สุด

# Longest-Job-First (LJF)

- เลือก Process ที่มี burst time สูงสุดมาทำก่อน
- จะเลือก  $P1 \rightarrow P2 \rightarrow P3$ 
  - $P1$ : burst time = 5s
  - $P2$ : burst time = 3s
  - $P3$ : burst time = 2s
- Waiting time  $P1 = 0$ ,  $P2 = 5$ ,  $P3 = 8$ : ดังนั้น  $0+5+8 = 13$
- Average waiting time =  $13/3 = 4.33s$

# Shortest-Job-First (SJF)

- เลือก Process ที่มี burst time ต่ำสุดมาทำก่อน
- จะเลือก P3 -> P2 -> P1
  - P1: burst time = 5s
  - P2: burst time = 3s
  - P3: burst time = 2s
- Waiting time P3 = 0, P2 = 2, P1 = 5: ดังนั้น  $0+5+2 = 7$
- Average waiting time =  $7/3 = 2.33s$



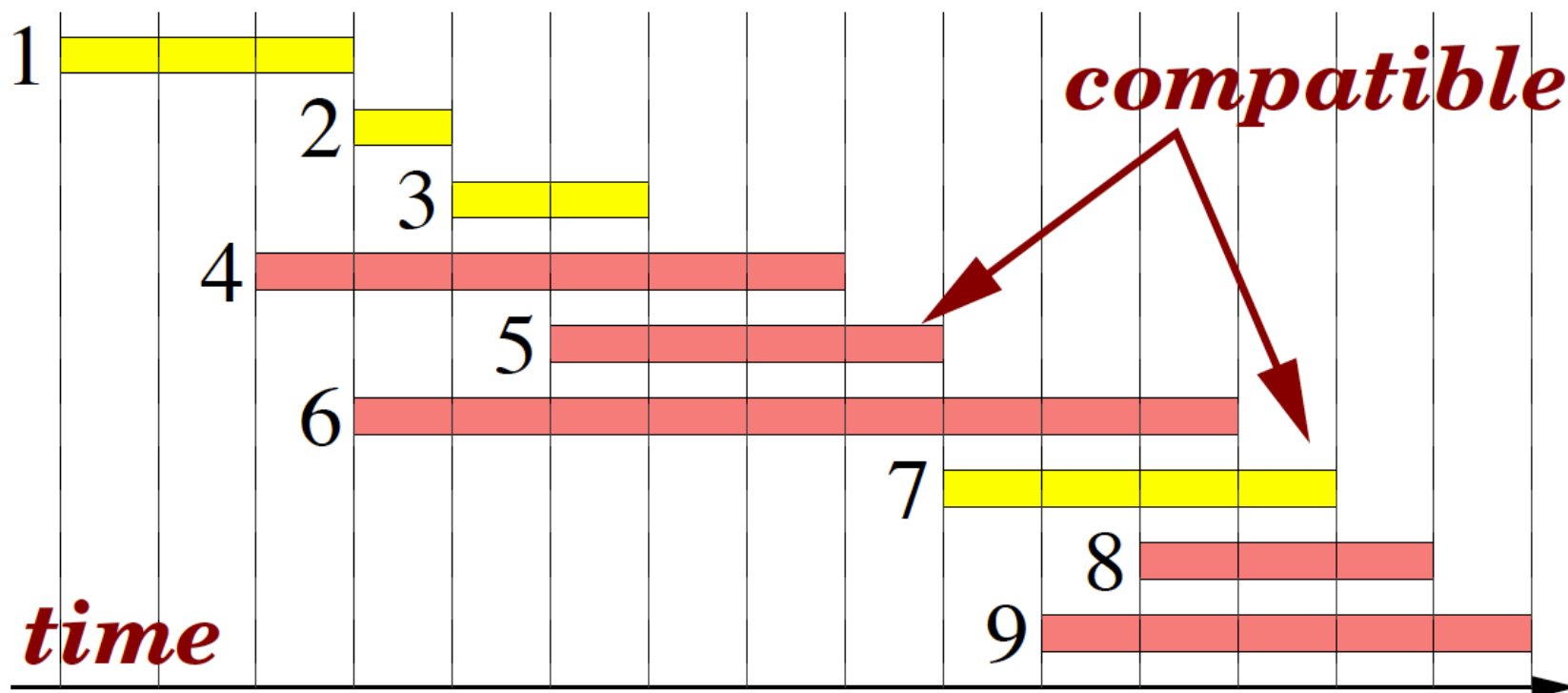
# พิสูจน์ว่า SJF เป็น optimal

- Waiting time ของ  $P_i$  จะเท่ากับผลรวมของ burst time ของ process ก่อนหน้า  $P_i = \sum(P_1 - P_{i-1})$
- Minimum sum = minimum sum of burst rates
- ดังนั้น SJF ให้คำตอบเป็น optimal

# Activity Selection problem

- สมมติมีห้องว่างสำหรับทำกิจกรรม 1 ห้อง
- มีกิจกรรมให้เลือกทำโดยที่  $s(\text{start})$  คือเวลาเริ่มต้น และ  $f(\text{finish})$  คือเวลาสิ้นสุด โดยจะเขียนด้วยสัญลักษณ์  $[s, f)$
- ตัวอย่างเช่น  $[3, 5)$  หมายถึง กิจกรรมเริ่มที่เวลา 3 และสิ้นสุด ณ เวลา 5
- กิจกรรมที่มีเวลาทับซ้อนกัน เช่น  $[3, 5)$  และ  $[4, 7)$  ไม่สามารถ
- โจทย์นี้ให้เลือกจัดกิจกรรมโดยใช้ห้องเดียวกัน ทำอย่างไรให้สามารถจัดกิจกรรมได้เยอะที่สุด

# Activity Selection problem



# แนวทางการเลือก

- เลือกกิจกรรมที่เริ่มเร็วที่สุด
- เลือกกิจกรรมที่ใช้เวลาน้อยที่สุด
- เลือกกิจกรรมที่จบเร็วที่สุด

## วิธีที่ 1: เลือกกิจกรรมที่เริ่มก่อน

- สมมติมีทั้งหมด 3 กิจกรรม  $[2,3)$   $[4,5)$   $[0,5)$  ห้องใช้ได้ 5 ชั่วโมง 0-5
- ทดสอบเลือกกิจกรรมที่เริ่มเร็วสุดมาทำก่อน
- โปรแกรมจะเลือก  $[0,5)$  ก่อน ซึ่งทำให้จัดกิจกรรมได้เพียงกิจกรรมเดียว
- ดังนั้นวิธีนี้ไม่ให้คำตอบที่ถูกต้อง

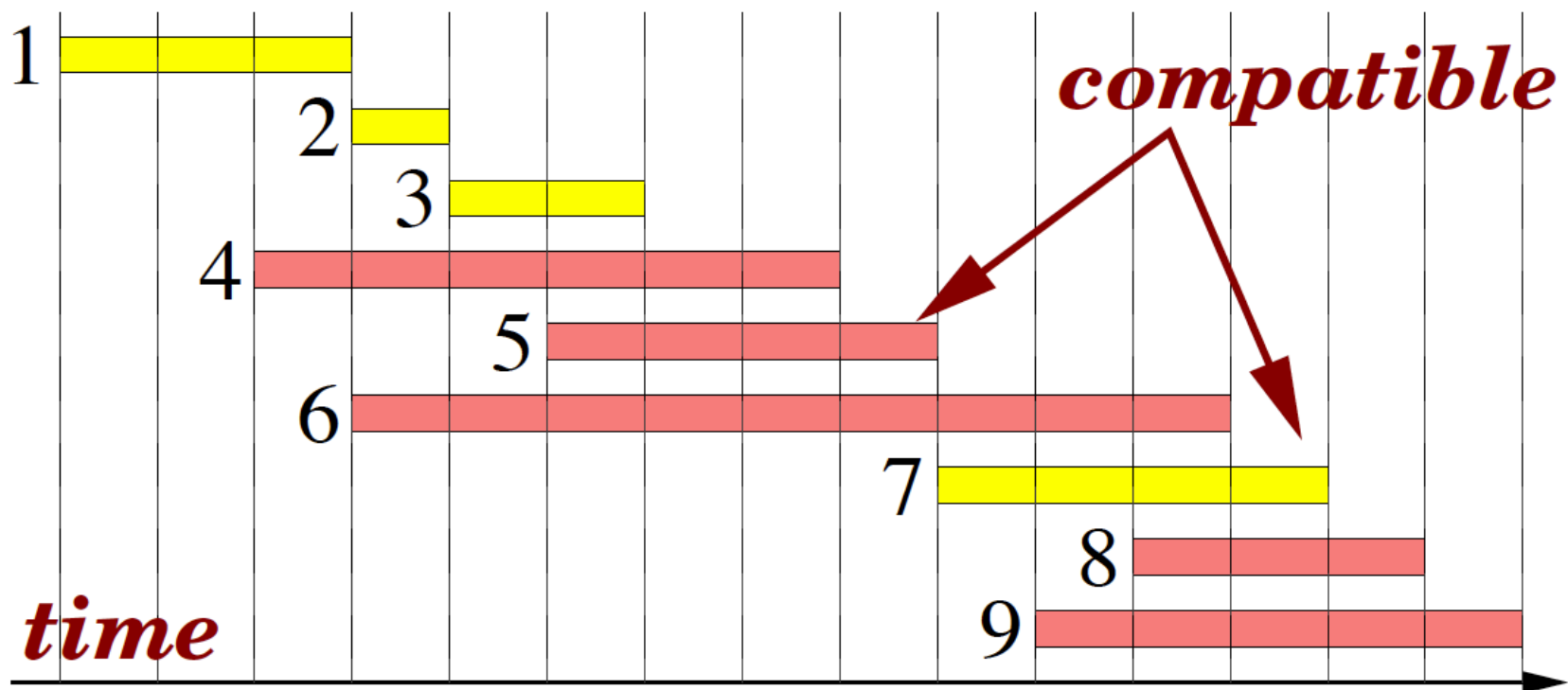
## วิธีที่ 2: เลือกกิจกรรมที่ใช้เวลาน้อยที่สุด

- สมมติมีทั้งหมด 3 กิจกรรม  $[0, 5)$ ,  $[4, 6)$ ,  $[6, 10)$  มีเวลาให้ 10 ชั่วโมง
- เลือกกิจกรรมที่ใช้เวลาน้อยที่สุดก่อน
- เลือก  $[4,6)$  มาทำก่อน
- จะเห็นได้ว่าเลือกกิจกรรมที่สั้นที่สุด ทำได้เพียงแค่งิจกรรมเดียว
- ไม่ให้คำตอบที่ถูกต้องที่สุด

## วิธีที่ 3: เลือกกิจกรรมที่เสร็จเร็วที่สุด

- สมมติให้มี 5 กิจกรรม  $[0, 6)$ ,  $[2, 3)$ ,  $[2, 4)$ ,  $[3, 5)$ ,  $[4, 5)$
- เลือกกิจกรรมที่เสร็จเร็วที่สุด  $f$  น้อยสุด
- เลือก  $[2,3)$  ก่อน หลังจากนั้นเลือก  $[4,5)$
- ทำได้สองกิจกรรม
- เป็นคำตอบที่ optimal

พิสูจน์ว่า วิธีที่ 3 ให้คำตอบ optimal





# Character Encoding

- โดยปกติแล้ว เวลาที่คอมพิวเตอร์เก็บข้อมูลตัวอักษร เช่น A B C จะเก็บอยู่ในรูปของ ASCII นั่นคือ จะใช้ 8 bit ในการแทนค่าแต่ละตัวอักษร
- ดังนั้น
  - A มีค่าเท่ากับ 01000001 (65)
  - Z มีค่าเท่ากับ 01011010 (90)
- สมมติ text file มีข้อมูลอักษรเป็น AAZAAAA ก็จะใช้พื้นที่เก็บ  $8 \times 8 = 64$  bits
- การเก็บข้อมูลตัวอักษรแบบนี้เรียกว่าเป็นการเก็บแบบ fixed-length

# Huffman Coding

- เนื่องจากความถี่ที่พบตัวอักษรแต่ละตัวใน text file นั้นไม่เท่ากัน
- จะเห็นได้ว่า A ซึ่งเป็นสระมักจะพบได้บ่อยกว่า Z ซึ่งเป็นอักษรที่ไม่ค่อยปรากฏในคำทั่วไป
- ดังนั้นหากเราสามารถใช้จำนวน bit ที่น้อยกว่าในการเก็บตัวอักษร A ก็จะสามารถประหยัดพื้นที่ในการเก็บข้อมูลได้
- ตัวอย่างเช่น หากให้  $A = 0$  และ  $Z = 11111111$  การเก็บข้อมูลของ string AAAAZAAA ก็จะใช้พื้นที่เพียง  $7 + 8 = 15$  bits

# Huffman Coding (ต่อ)

- ข้อดีของการเก็บอักขระแบบ fixed-length นั้นคือช่วยให้โปรแกรมสามารถแยกแยะได้ว่า bit ไหนคือ bit เริ่มต้นของแต่ละอักขระ
- ในทางกลับกันหากเราใช้ variable-length ที่ไม่ถูกต้องอาจจะเกิดปัญหาการแปลงข้อมูลกลับ
- ตัวอย่างเช่น หากให้  $A = 1$ ,  $B = 11$  เมื่อ  $AB$  อยู่ติดกันจะได้สายอักขระเป็น 111 ซึ่งหากต้องการจะแปลงกลับก็ไม่แน่ใจว่าจะได้  $AB$  หรือ  $BA$

# Huffman Coding (ต่อ)

- ดังนั้นวิธีการแก้ปัญหาคือ จำนวน bit ที่แทนค่าอักขระของแต่ละตัวอักษรห้ามเป็น prefix ของตัวอักษรอีกตัว
- ตัวอย่างเช่น หาก  $A = 010$  ตัวอักษรใด ๆ ก็ตามห้ามมีค่าเป็น 0 หรือ 01
- ทดสอบทฤษฎี, สมมติ  $A = 010$ ,  $B = 0$ ,  $C = 01$
- $ABC = 010001$  หากต้องการแปลงกลับ ไม่แน่ใจว่าตัวอักษรแรกจะเป็นตัวไหน

# Huffman Coding (ต่อ)

- จุดมุ่งหมายของการสร้าง Huffman coding คือการหาวิธีที่จะใช้จำนวน bit ที่น้อยที่สุดในการแทนค่าตัวอักษรที่พบบ่อยที่สุด
- สมมติมีอักษรทั้งหมด 5 ตัวเท่านั้นในสายอักขระคือ A,B,C,D,E
- จำนวนความถี่ที่พบคือ  $A=30, B=20, C=25, D=28, E=40$
- คำถาม จะแทนตัวอักษรแต่ละตัวด้วยกี่ bit และให้ตัวไหนแทนด้วย bit แบบไหนถึงจะใช้พื้นที่น้อยที่สุดในการเก็บข้อมูล

# วิธีการแทนค่าอักษร Huffman Coding

- จะแทนที่อย่างไรดี?
- กฎ
  - แทนที่อักษรที่พบบ่อยที่สุดด้วยจำนวน bit ที่น้อยที่สุด
  - ห้ามอักษรหนึ่งตัวเป็น prefix ของอักษรตัวอื่น

# วิธีการใช้ tree เพื่อหา Huffman Coding

- เรียงลำดับ frequency จะได้ว่า E,A,D,C,B

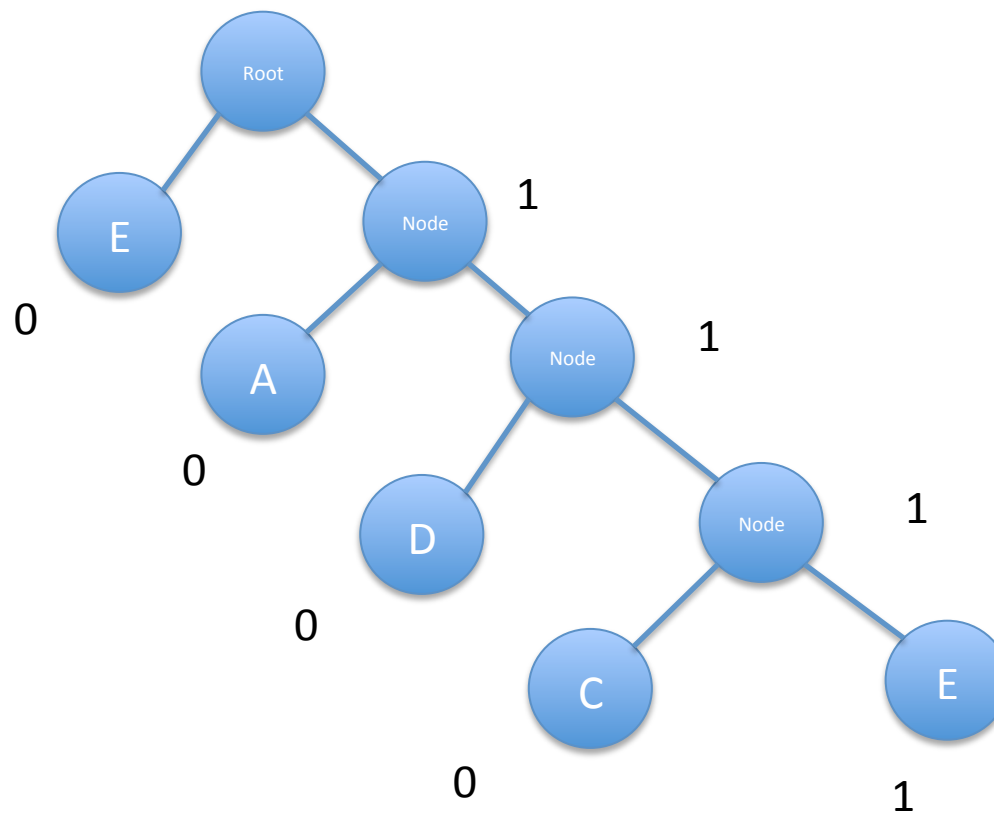
— E = 0

— A = 10

— D = 110

— C = 1110

— E = 1111



Time to Code



# 1. โปรแกรมแลกเหรียญ

- ให้นักเรียนเขียนโปรแกรมที่รับแลกเหรียญ โดยโปรแกรมจะมีเหรียญทั้งหมด 5 ชนิดคือ 1,2,5,10 และ 20 บาท
- โปรแกรมของนักเรียนจะรับ input เป็นจำนวนเงินที่ต้องการแลก และจะ output เป็นจำนวนเหรียญแต่ละชนิดออกมา
- ดูตัวอย่าง input output ได้ที่ folder Q1 Coin

## 2. แก้ปัญหา Fractional Knapsack

- ให้นักเรียนเขียนโปรแกรมในการประมวลผลปัญหา Fractional Knapsack โดยให้รับ input และ output ดังนี้
- Input
  - รับจำนวนกองของรางวัล เช่น 5 กอง, 10 กอง
  - รับข้อมูลของกองของรางวัลและมูลค่าของกอง เช่น 200 400 แปลว่า กองของรางวัลมีน้ำหนัก 200 และมีมูลค่า 400
  - รับความจุของกระเป๋า เป็นจำนวนเต็ม เช่น 200
- Output
  - คำนวนและแสดงผลว่า จะเลือกของจากกองไหน เป็นจำนวนเท่าไร
  - ตัวอย่าง input output ได้จาก Q2 Knapsack

### 3. โปรแกรมตรวจสอบ string

- นักวิจัยท่านหนึ่งได้เขียนโปรแกรมในการเข้ารหัสสายอักขระโดยทำการแทรก random string ไประหว่างตัวอักษร ให้นักเรียนเขียนโปรแกรมตรวจสอบ string ว่าเป็น string ที่ได้รับการเข้ารหัสหรือไม่
- ดูตัวอย่าง input output ได้ที่ Q3 Encryption
- ตัวอย่างเช่น
  - sequence subsequence -> Yes
  - person compression -> No
  - VERDI vivaVittorioEmanueleReDItalia -> Yes
  - caseDoesMatter CaseDoesMatter -> No

## 4. เลือกกิจกรรม

- มีห้องว่างให้ใช้งานอยู่ 1 ห้อง ให้นักเรียนเขียนโปรแกรมที่รับจำนวนและรายละเอียดของกิจกรรม เวลาเริ่ม  $s$  และระยะเวลาของกิจกรรม  $t$  โดยที่โปรแกรมจะคำนวณและแสดงผลลำดับและจำนวนกิจกรรมที่ได้รับเลือกให้จัด
- ดูตัวอย่าง input output ได้ที่ Q4 Activity Selection
- ตัวอย่าง input
  - จำนวนกิจกรรมที่จัด
  - เวลาเริ่ม และเวลาสิ้นสุดของแต่ละกิจกรรม เช่น 0 5 (เริ่ม 0, ใช้เวลา 5)

## 5. ปัญหาถังขยะ

- ให้นักเรียนเขียนโปรแกรมหาจำนวนถังขยะที่น้อยที่สุดที่ต้องใช้ในการใส่ขยะ โดยขยะแต่ละชิ้นจะมีน้ำหนักที่แตกต่างกันออกไป
- ดูตัวอย่าง input output ได้ที่ Q5 Minimum bins
- ตัวอย่าง input output
  - จำนวนขยะ 5
  - น้ำหนักของขยะ 7,2,3,8,1,9
  - ขนาดของถังขยะ เช่น 10
- ตัวอย่าง output
  - 3

## 6. Egyptian Fraction

- Egyptian Fraction เป็นการเขียนเศษส่วนให้อยู่ในรูปที่มีส่วนเท่ากับ 1
- ตัวอย่างเช่น
  - $2/3 = 1/2 + 1/6$
  - $6/14 = 1/3 + 1/11 + 1/231$
  - $12/13$  is  $1/2 + 1/3 + 1/12 + 1/156$
- จงเขียนโปรแกรมที่ใช้ในการคำนวณหา Egyptian Fraction ของเศษส่วนที่ได้รับเข้าไปในแบบ
- ดูตัวอย่าง input/output ได้ที่ Q7 Egyptian