COMP319 Algorithms 1
Lecture 12
Greedy Algorithms

Instructor: Gil-Jin Jang

Greedy Algorithms

Slide credit:

https://prof.ysu.ac.kr/, 영산대학교

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Table of Contents

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

그리디 (Greedy) 알고리즘

- 그리디 알고리즘은 최적화 문제를 해결
- **최적화 (optimization) 문제**

→가능한 해들 중 가장 좋은 (최대 or 최소) 해 찾는 문제

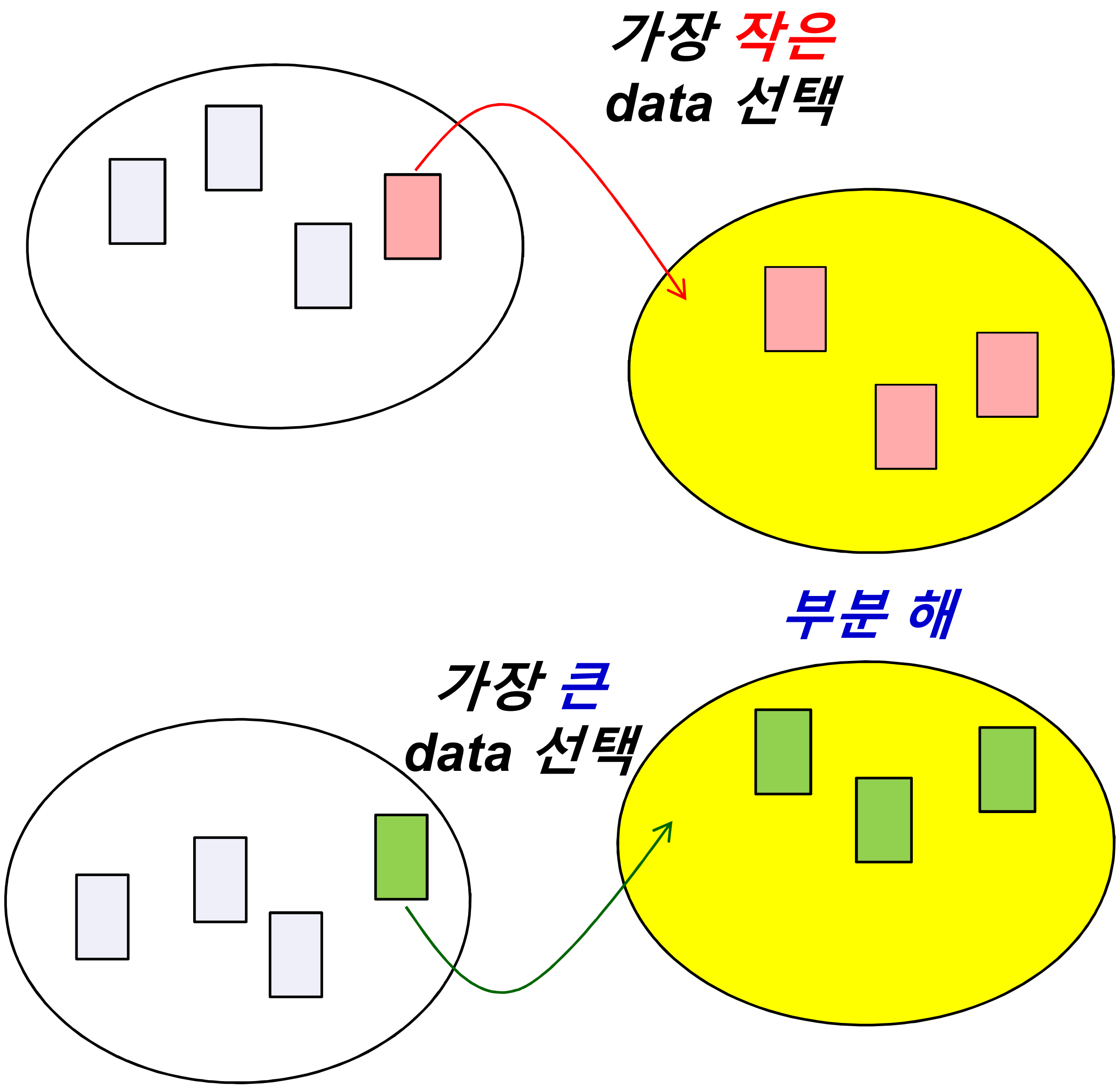
- 욕심쟁이 방법, 탐욕적 방법, 탐욕 알고리즘 등
- **그리디 알고리즘**
  - 입력 데이터 간의 관계를 고려하지 않고 수행 과정에서 '**욕심내어**' **최소값 또는 최대값을 가진 데이터를 선택**

- 이런 선택을 '**근시안적**'**인 선택**이라고 말하기도 함

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

그리디 알고리즘 수행 과정

- **그리디알고리즘**은 근시안적인 선택으로 **부분적인 최적해를 찾고,** **이들을 모아서 문제의 최적해**를 얻는다



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 그리디 알고리즘은 **일단 한번 선택하면, 이를 절대로 번복하지 않는다.**
- 즉, 선택한 데이터를 버리고 다른 것을 취하지 않는다.
- **매우 단순하며, 또한 제한적인 문제들만이** 그리디 알고리즘으로 해결된다.
- 8장에서 다루는 근사 알고리즘
- 9장의 해를 탐색하는 기법들 중의 하나인 분기 한정 기법

4.1 동전 거스름돈

- 동전 거스름돈 (Coin Change) 문제
- 남은 액수를 초과하지 않는 조건하에 **'욕심내어'** **가장 큰 액면의 동전을 취하는 것**

- 동전 거스름돈 문제의 **최소 동전 수를 찾는 그리디 알고리즘**
- 단, 동전의 액면은 500원, 100원, 50원, 10원, 1원

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

CoinChange

입력: 거스름돈 액수 W
출력: 거스름돈 액수에 대한 최소 동전 수

```
1. change=W,   n500=n100=n50=n10=n1=0
                         // n500, n100, n50, n10, n1은 각각의 동전 카운트
2. while ( change ≥ 500 )
      change = change-500, n500++      // 500원짜리 동전 수를 1 증가
3. while ( change ≥ 100 )
      change = change-100, n100++      // 100원짜리 동전 수를 1 증가
4. while ( change ≥ 50 )
      change = change-50, n50++        // 50원짜리 동전 수를 1 증가
5. while ( change ≥ 10 )
      change = change-10, n10++        // 10원짜리 동전 수를 1 증가
6. while ( change ≥ 1 )
      change = change-1, n1++          // 1원짜리 동전 수를 1 증가
7. return (n500+n100+n50+n10+n1) // 총 동전 수를 리턴한다.
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- Line 1: change를 입력인 거스름돈 액수 W로 놓고, 각 동전 카운트를 n500 = n100 = n50 = n10 = n1 = 0으로 초기화
- Line 2~6: 차례로 500원, 100원, 50원, 10원, 1원짜리 동전을 각각의 while-루프를 통해 현재 남은 거스름돈 액수인 change를 넘지 않는 한 계속해서 같은 동전으로 거슬러 주고, 그 때마다 각각의 동전 카운트를 1 증가시킨다.
- Line 7에서는 동전 카운트들의 합을 리턴한다.

- CoinChange 알고리즘은 남아있는 거스름돈인 change에 대해 가장 높은 액면의 동전을 거스르며, 500원짜리 동전을 처리하는 line 2에서는 100원짜리, 50원짜리, 10원짜리, 1원짜리 동전을 몇 개씩 거슬러 주어야 할 것인지에 대해서는 전혀 고려하지 않는다.

- **이것이 바로 그리디 알고리즘의 근시안적인 특성**

- 거스름돈 760원, CoinChange 알고리즘 수행 과정

- Line 1: change = 760, n500 = n100 = n50= n10 = n1 = 0 으로 초기화
- **Line 2: change가 500보다 크므로 while-조건이 '참', change = change-500 = 760-500 = 260이고, n500=1**
- change가 500보다 작으므로 line 2의 while-루프는 더 이상 수행되지 않음

- **Line 3: change> 100, while-조건이** '참'이 되어 change = change-100 = 260-100 = 160이고, **n100 = 1**
- 다음도 change> 100이므로 while-조건이 역시 '참', change = change-100 = 160-100 = 60이고, **n100=2**

- 그러나 그 다음엔 change가 60이므로 100보다 작아서 while-루프는 수행되지 않는다

- **Line 4: change> 50이므로** while-조건이 ‘참’, change = change-50 = 60-50 = 10이 되고, n50=1

- 다음은 change가 50보다 작으므로 while-루프는 수행되지 않는다.

- **Line 5: change>10**이므로 while-조건이 ‘참’, change = change-10 = 10-10 = 0이 되고, n10=1

- **Line 6: change=0이므로** while-조건이 ‘거짓’이 되어 while-루프는 수행되지 않는다.

- **Line 7에서는 n500+n100+n50+n10+n1 = 1+2+1+1+0 = 5**를 리턴

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 만일 **160원짜리 동전을 추가로 발행한다면**, CoinChange 알고리즘이 항상 최소 동전 수를 계산할 수 있을까?
- 거스름돈이 200원, CoinChange 알고리즘은 **160원짜리 동전 1개와 10원짜리 동전 4개로 총 5개 리턴**
- **CoinChange 알고리즘은 항상 최적의 답을 주지 못한다.**
- 5 장에서는 어떤 경우에도 최적해를 찾는 동전 거스름돈을 위한 동적 계획 알고리즘을 소개한다.

CoinChange 알고리즘의 결과

최소 동전의 거스름돈

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

4.4 부분 배낭 문제

- **n개의 물건이 있고, 각 물건은 무게와 가치**를 가지고 있으며,
- **배낭이 한정된 무게의 물건들을 담을 수 있을 때, 최대의 가치를 갖도록 배낭에 넣을 물건들을 정하는 문제**
- 원래 배낭 문제는 물건을 통째로 배낭에 넣어야 되지만, **부분 배낭 (Fractional Knapsack) 문제는 물건을 부분적으로 담는 것을 허용**

- **부분 배낭 문제에서는** 물건을 부분적으로 배낭에 담을 수 있으므로, **최적해를 위해서 '욕심을 내어' 단위 무게 당 가장 값나가는 물건을 배낭에 넣고**, 계속해서 그 다음으로 값나가는 물건을 넣는다.
- 그런데 만일 그 다음으로 값나가는 물건을 '통째로' 배낭에 넣을 수 없게 되면, **배낭에 넣을 수 있을 만큼만 물건을 부분적으로 배낭에 담는다.**

FractionalKnapsack

입력: n개의 물건, 각 물건의 무게와 가치, 배낭의 용량 C

출력: 배낭에 담은 물건 리스트 L, 배낭에 담은 물건의 가치 합 v

1. 각 물건에 대해 **단위 무게 당 가치를 계산**한다.
2. 물건들을 **단위 무게 당 가치를 기준으로 내림차순으로 정렬**하고, **정렬된 물건 리스트를 S**라고 하자.
3. **L=∅, w=0, v=0**

   // L은 배낭에 담을 **물건 리스트, w는 배낭에 담긴 물건들의 무게의 합, v**는 배낭에 담긴 물건들의 **가치의 합**
4. **S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져온다.**

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

```
5. while ( (w+x의 무게) ≤ C ) {
6.        x를 L에 추가시킨다.
7.        w = w +x의 무게
8.        v = v +x의 가치
9.        x를 S에서 제거한다.
10.       S에서 단위 무게 당 가치가 가장 큰 물건 x를 가져
          온다.
   }
11. if ((C-w) > 0) { // 배낭에 물건을 부분적으로 담을 여유가 있으면
12.       물건 x를 (C-w)만큼만 L에 추가한다.
13.       v = v +(C-w)만큼의 x의 가치
    }
14. return L, v
```

- Line 1~2: 각 물건의 **단위 무게 당 가치를 계산**하여, 이를 기준으로 물건들을 **내림차순으로 정렬**한다.
- Line 5~10의 while-루프를 통해서 다음으로 **단위 무게 당 값나가는 물건을 가져다 배낭에 담고**, 만일 가져온 물건을 배낭에 담을 경우 **배낭의 용량이 초과되면**, (즉, while-루프의 조건이 ‘거짓’이 되면) 가져온 물건을 ‘통째로’ 담을 수 없게 되어 **루프를 종료한다**.
- Line 11: 현재까지 배낭에 담은 **물건들의 무게 w가 배낭의 용량 C 보다 작으면**, (즉, if-조건이 ‘참’이면) line 12~13에서 **해당 물건을 (C-w)만큼만 배낭에 담고, (C-w)만큼의 x의 가치를 증가시킨다**.
- Line 14: 최종적으로 배낭에 담긴 **물건들의 리스트 L과 배낭에 담긴 물건들의 가치 합 v를 리턴한다**.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 4개의 금속 분말
- 배낭의 최대 용량이 40그램일 때, FractionalKnapsack 알고리즘의 수행 과정
- Line 1~2의 결과: S=[백금, 금, 은, 주석]

물건	단위 그램당 가치
백금	6만원
금	5만원
은	4천원
주석	1천원

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- Line 3: L=∅, w=0, v=0로 각각 초기화한다.
- Line 4: **S=[백금, 금, 은, 주석]로부터** **백금**을 가져온다.
- Line 5: while-루프의 조건 **((w+백금의 무게) ≤ C) = ((0+10)<40)이 '참'이다.**
- Line 6: **백금을 배낭 L에 추가**시킨다. 즉, **L=[백금]**이 된다.
- Line 7: **w = w(백금의 무게) = 0+10g = 10g**
- Line 8: **v = v(백금의 가치) = 0+60만원 = 60만원**
- Line 9: S에서 백금을 제거한다. **S=[금, 은, 주석]**
- Line 10: **S에서** **금**을 가져온다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- Line 5: while-루프의 조건 ((**w+금의 무게) ≤ C**) = **((10+15)<40)이 '참'**이다.
- Line 6: 금을 배낭 L에 추가시킨다. **L=[백금, 금]**
- Line 7: **w** = w+(금의 무게) = **10g+15g = 25g**
- Line 8: **v** = v+(금의 가치) = **60만원+75만원 = 135만원**
- Line 9: S에서 금을 제거한다. **S=[은, 주석]**
- Line 10: S에서 은을 가져온다.
- Line 5: while-루프의 조건 (**(w+은의 무게) ≤ C) = ((25+25)<40)이 '거짓'이므로 루프를 종료**한다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- Line 11: if-조건 ((**C-w) >0**)이 '참'이다. 즉, **40-25 = 15 > 0**이기 때문이다.
- Line 12: 따라서 **은을 C-w=(40-25)=15g만큼만** 배낭 L에 추가시킨다.
- Line 13: **v = v+(15g x 4천원/g) = 135만원+6만원 = 141만원**
- Line 14: 배낭 **L=[백금 10g, 금 15g, 은 15g]**과 **가치의 합 v = 141만원**을 리턴한다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

시간복잡도

- Line 1: n개의 물건 각각의 **단위 무게 당 가치를 계산하는 데는** **O(n)** 시간 걸리고, line 2에서 물건의 단위 무게 당 가치에 대해서 **내림차순으로 정렬하기 위해** **O(nlogn)** 시간이 걸린다.
- Line 5~10의 while-루프의 수행은 n번을 넘지 않으며, **루프 내부의 수행은 O(1)** 시간이 걸린다. 또한 line 11~14도 **각각 O(1)** 시간 걸린다.
- 따라서 알고리즘의 시간복잡도는 **O(n)+O(nlogn)+nxO(1)+O(1) = O(nlogn)**이다.

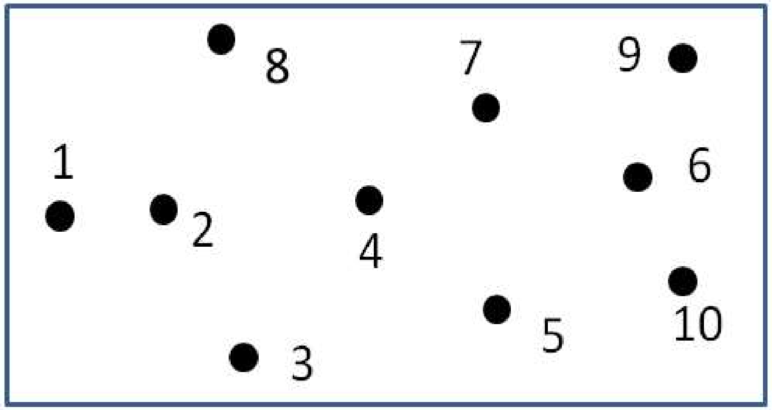
응 용

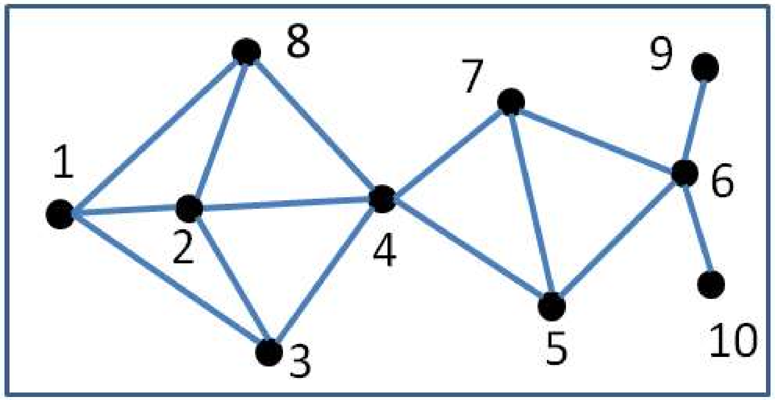
- 0-1 배낭 문제는 최소의 비용으로 자원을 할당하는 문제로서, **조합론, 계산이론, 암호학**, 응용수학 분야에서 기본적인 문제로 다루어진다.
- 응용 사례로는 '버리는 부분 최소화시키는' **원자재 자르기 (Raw Material Cutting),**
- 자산투자 및 금융 포트폴리오 (Financial Portfolio)에서의 최선의 선택
- Merkle–Hellman 배낭 암호 시스템의 키 (Key) 생성에도 활용된다.

4.5 집합 커버 문제

- **n개의 원소를 가진 집합인 U**가 있고, **U의 부분 집합들을 원소로 하는 집합 F**가 주어질 때,
- F의 원소들인 집합들 중에서 **어떤 집합들을 선택하여 합집합하면 U와 같게** 되는가?
- 집합 커버 (cover) 문제는 **F에서 선택하는 집합들의 수를 최소화하는 문제**이다.

- **예제: 신도시 계획 학교 배치**
- **10개의 마을이 신도시**에 있다.
- 이때 아래의 **2가지 조건이 만족**되도록 학교의 위치를 선정하여야 한다고 가정하자.
  - **학교는 마을에 위치해야 한다.**
  - **등교 거리는 걸어서 15분 이내이어야 한다.**

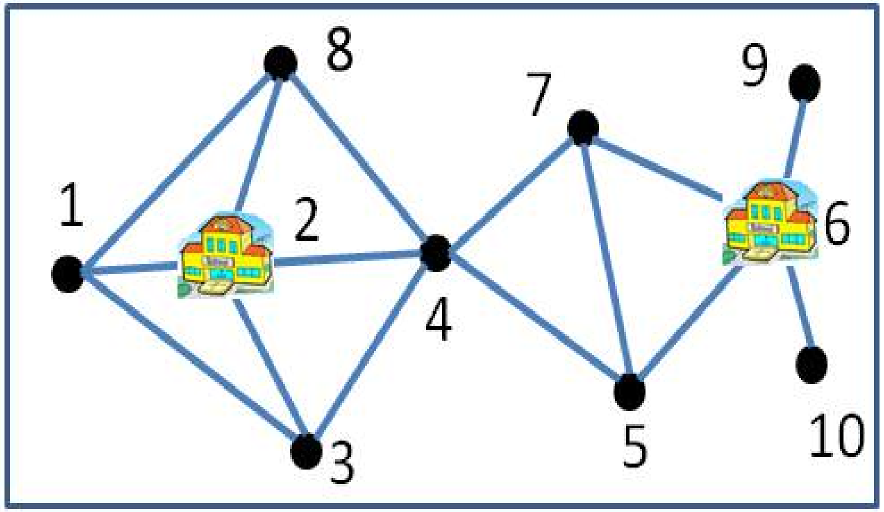




등교 거리가 15분 이내인 마을 간의 관계

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 어느 마을에 학교를 **신설해야 학교의 수가 최소**로 되는가?
- **2번 마을에 학교를 만들면 마을 1, 2, 3, 4, 8의 학생들이 15분 이내에 등교**할 수 있고 (즉, 마을 1, 2, 3, 4, 8이 '커버'되고),
- **6번 마을에 학교를 만들면 마을 5, 6, 7, 9, 10이 커버**된다.
- 즉, **2번과 6번이 최소**이다.



- **신도시 계획 문제를 집합 커버 문제로 변환:**
- **S_i는 마을 i에 학교를 배치했을 때 커버되는 마을의 집합**

  U={1, 2, 3, 4, 5, 6, 7, 8, 9, 10} // 신도시의 마을 10개

  F={S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}}

  S_1={1, 2, 3, 8} **S_5={4, 5, 6, 7}** **S_9={6, 9}**

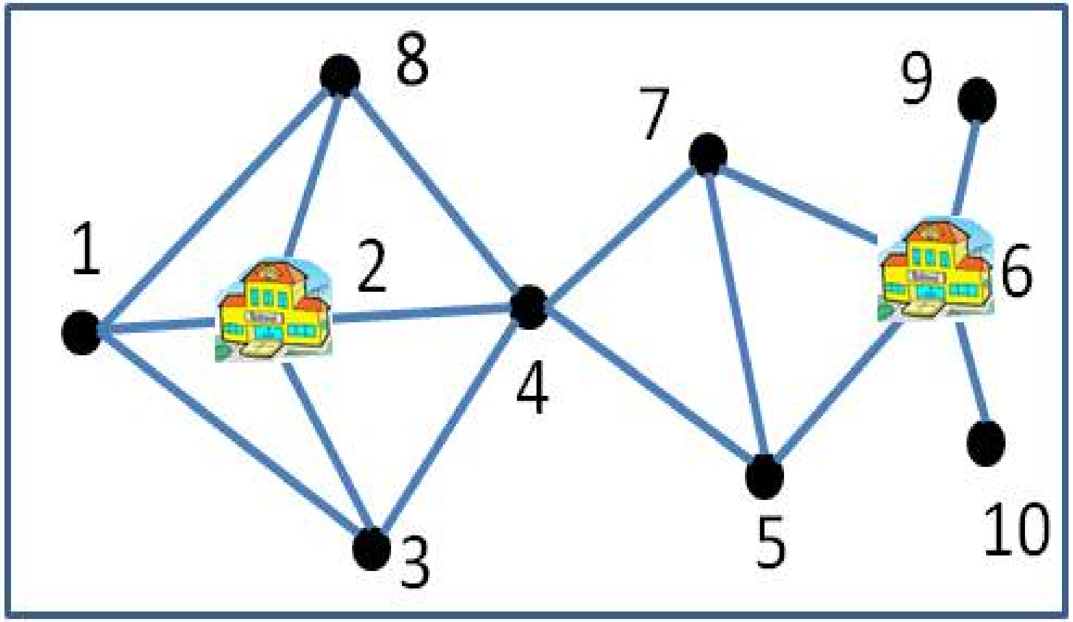
  S_2={1, 2, 3, 4, 8} **S_6={5, 6, 7, 9, 10}** **S_{10}={6, 10}**

  S_3={1, 2, 3, 4} **S_7={4, 5, 6, 7}**

  S_4={2, 3, 4, 5, 7, 8} **S_8={1, 2, 4, 8}**
- **S_i 집합들 중에서 어떤 집합들을 선택하여야 그들의 합집합이 U와 같은가?** 단, 선택된 집합의 수는 최소이어야 한다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 이 문제의 답은 $S_2 \cup S_6 = \{1, 2, 3, 4, 8\} \cup \{5, 6, 7, 9, 10\} = \{1, 2, 3, 4, 5, 6, 7, 8\} = U$이다.



- 집합 커버 문제의 최적해는 어떻게 찾아야 할까?
- **F에 n개의 집합들이 있다고 가정**해보자.
- 가장 단순한 방법은 F에 있는 집합들의 **모든 조합을 1개씩 합집합하여 U가 되는지 확인하고, U가 되는 조합의 집합 수가 최소인 것을 찾는 것**이다.
- 예를 들면, **F={S_1, S_2, S_3}일 경우**, 모든 조합이란, **S_1, S_2, S_3, $S_1 \cup S_2$, $S_1 \cup S_3$, $S_2 \cup S_3$, $S_1 \cup S_2 \cup S_3$**이다.
  - 집합이 1개인 경우 3개 = ${}_5C_1$,
  - 집합이 2개인 경우 3개 = ${}_3C_2$,
  - 집합이 3개인 경우 1개 = ${}_3C_3$이다.
  - **총합은 3+3+1= 7 = 2^3-1 개이다**.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- **n개의 원소가 있으면 (2^n-1)개를 다 검사하여야** 하고, **n이 커지면 최적해를 찾는 것은 실질적으로 불가능하다**.
- 이를 극복하기 위해서는 **최적해를 찾는 대신에 최적해에 근접한 근사해 (approximation solution)를 찾는 것이다.**

집합 커버 알고리즘

SetCover

입력: U, F={S_i}, i=1,⋯,n

출력: 집합 커버 C

1. C=∅
2. while (U≠∅) do {
3. **U의 원소들을 가장 많이 포함하고 있는 집합 S_i를 F에서 선택한다.**
4. **U=U-S_i**
5. **S_i를 F에서 제거하고, S_i를 C에 추가한다.**

   }
6. return C

- Line 1: C를 공집합으로 초기화시킨다.
- Line 2~5의 while-루프에서는 **집합 U가 공집합이 될 때까지 수행된다**.
- Line 3: '그리디'하게 **U와 가장 많은 수의 원소들을 공유하는 집합 S_i를 선택한다**.
- Line 4: **S_i의 원소들을 U에서 제거한다**. 왜냐하면 S_i의 원소들은 커버된 것이기 때문이다. 따라서 U는 아직 커버되지 않은 원소들의 집합이다.
- Line 5: **S_i를 F로부터 제거하여, S_i가 line 3에서 더 이상 고려되지 않도록 하며, S_i를 집합 커버 C에 추가한다**.
- Line 6: C를 리턴한다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

SetCover 알고리즘의 수행 과정

U={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

F={S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}}

S_1={1, 2, 3, 8}

S_2={1, 2, 3, 4, 8}

S_3={1, 2, 3, 4}

S_4={2, 3, 4, 5, 7, 8}

S_5={4, 5, 6, 7}

S_6={5, 6, 7, 9, 10}

S_7={4, 5, 6, 7}

S_8={1, 2, 4, 8}

S_9={6, 9}

S_{10}={6, 10}

- Line 1: C=∅로 초기화
- Line 2: while-조건 **(U≠∅)=({1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ≠ ∅)**이 '참'이다.
- Line 3: U의 원소를 가장 많이 커버하는 집합인 **S_4={ 2, 3, 4, 5, 7, 8}을 F에서 선택한다**.
- Line 4: **$U = U - S_4 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} - \{2, 3, 4, 5, 7, 8\} = \{1, 6, 9, 10\}$**
- Line 5: S_4를 **F에서 제거하고, 즉, $F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_4\} = \{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\}$**가 되고, **$S_4$를 C에 추가한다. 즉, $C = \{S_4\}$이다.**

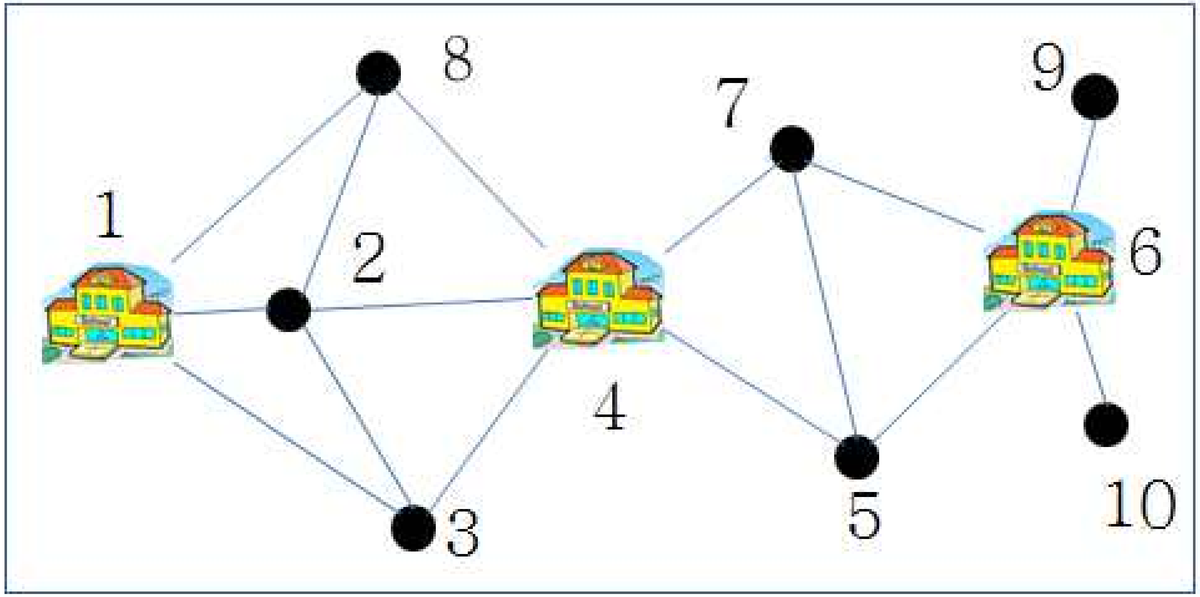
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- Line 2: while-조건 **($U \neq \emptyset$) = ($\{1, 6, 9, 10\} \neq \emptyset$)**이 '참'이다.
- Line 3: U의 원소를 가장 많이 커버하는 집합인 **S_6={5, 6, 7, 9, 10}을 F에서 선택한다.**
- Line 4: **$U = U - S_4 = \{1, 6, 9, 10\} - \{5, 6, 7, 9, 10\} = \{1\}$**
- Line 5: **S_6을 F에서 제거하고, 즉, $F=\{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_6\} = \{S_1, S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\}$**이 되고, **$S_6$을 C에 추가한다. 즉, $C = \{S_4, S_6\}$이다.**

- Line 2: while-조건 **($U \neq \emptyset$) = ($\{1\} \neq \emptyset$)**이 '참'이다.
- Line 3.: U의 원소를 가장 많이 커버하는 집합인 **S_1= {1, 2, 3, 8}을 F에서 선택한다**. S_1 대신에 S_2, S_3, S_8 중에서 어느 하나를 선택해도 무방하다.
- Line 4: **$U = U - S_1 = \{1\} - \{1, 2, 3, 8\} = \emptyset$**
- Line 5: **S_1을 F에서 제거하고, 즉, $F=\{S_1, S_2, S_3, S_5, S_6, S_7, S_8, S_9, S_{10}\} - \{S_1\} = \{S_2, S_3, S_5, S_7, S_8, S_9, S_{10}\}$**이 되고, **$S_1$을 C에 추가한다. 즉, $C = \{S_1, S_4, S_6\}$이다**.
- Line 2: while-조건 ($U \neq \emptyset$) = ($\emptyset \neq \emptyset$)이 '거짓'이므로, 루프를 끝낸다.
- Line 6: **$C=\{S_1, S_4, S_6\}$을 리턴한다.**

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- **SetCover 알고리즘의 최종해**



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

시간복잡도

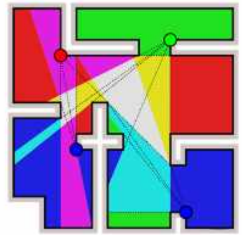
- 먼저 **while-루프가** 수행되는 횟수는 **최대 n번이다**. 왜냐하면 루프가 1번 수행될 때마다 집합 U의 원소 1개씩만 커버된다면, 최악의 경우 루프가 n번 수행되어야 하기 때문이다.
- **루프가 1번 수행될 때의 시간복잡도를 살펴보자**.
- Line 2의 while-루프 조건 (U≠∅)을 검사는 **O(1) 시간이** 걸린다. 왜냐하면 U의 현재 원소 수를 위한 변수를 두고 그 값이 0인지를 검사하면 되기 때문이다.

- Line 3: U의 원소들을 가장 많이 포함하고 있는 집합 S를 찾으려면, 현재 남아있는 S_i들 각각을 U와 비교하여야 한다. 따라서 **S_i들의 수는 최대 n이고, 각 S_i와 U의 비교는 O(n) 시간이 걸리므로, line 3은 $O(n^2)$ 시간이 걸린다.**
- Line 4: 집합 U에서 **집합 S_i의 원소를 제거하는 것이므로 O(n) 시간이** 걸린다.
- Line 5: S_i를 F에서 제거하고, S_i를 C에 추가하는 것이므로 O(1) 시간이 걸린다.
- 따라서 루프 1회의 시간복잡도는 **$O(1)+O(n^2)+O(n)+O(1) = O(n^2)$이다.**
- 따라서 **시간복잡도는 $O(n) \times O(n^2) = O(n^3)$이다.**

- **근사 알고리즘은 근사해가 최적해에 얼마나 근사한지 (즉, 최적해에 얼마나 가까운지)를 나타내는 근사 비율 (approximation ratio)을 알고리즘과 함께 제시하여야** 한다.
- SetCover 알고리즘의 근사 비율은 Klogn으로서, 그 의미는 SetCover 알고리즘의 최악 경우의 해일지라도 그 집합 수가 Klogn개를 넘지 않는다는 뜻이다. 여기서 K는 최적해의 집합의 수이다.
- 신도시 계획 예제에서는 최적해가 집합 2개로 모든 마을을 커버했으므로, Klogn = 2xlog10 < 2x4 = 8이다.
- 즉, SetCover 알고리즘이 찾는 근사해의 집합 수는 8개를 초과하지 않는다는 것이다.
- 집합 문제의 최적해를 찾는 데는 지수 시간이 걸리나, SetCover 알고리즘은 $O(n^3)$ 시간에 근사해를 찾으며 그 해도 실질적으로 최적해와 비슷하다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

응 용



- 도시 계획 (City Planning)에서 공공 기관 배치하기
- 경비 시스템: 미술관, 박물관, 기타 철저한 경비가 요구되는 장소 (Art Gallery 문제)의 CCTV 카메라의 최적 배치
- 컴퓨터 바이러스 찾기: 알려진 바이러스들을 '커버'하는 부분 스트링의 집합 - IBM에서 5,000개의 알려진 바이러스들에서 9,000개의 부분 스트링을 추출하였고, 이 부분 스트링의 집합 커버를 찾았는데, 총 180개의 부분 스트링이었다. 이 180개로 컴퓨터 바이러스의 존재를 확인하는데 성공하였다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 대기업의 구매 업체 선정: 미국의 자동차 회사인 GM은 부품 업체 선정에 있어서 각 업체가 제시하는 여러 종류의 부품들과 가격에 대해, 최소의 비용으로 구입하려고 하는 부품들을 모두 '커버'하는 업체를 찾기 위해 집합 문제의 해를 사용하였다.
- 기업의 경력 직원 고용: 예를 들어, 어느 IT 회사에서 경력 직원들을 고용하는데, 회사에서 필요로 하는 기술은 알고리즘, 컴파일러, 앱 (App) 개발, 게임 엔진, 3D 그래픽스, 소셜 네트워크 서비스, 모바일 컴퓨팅, 네트워크, 보안이고, 지원자들은 여러 개의 기술을 보유하고 있다. 이 회사가 모든 기술을 커버하는 최소 인원을 찾으려면, 집합 문제의 해를 사용하면 된다.
- 그 외에도 비행기 조종사 스케줄링 (Flight Crew Scheduling), 조립 라인 균형화 (Assembly Line Balancing), 정보 검색 (Information Retrieval) 등에 활용된다.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

4.6 작업 스케줄링

- 기계에서 수행되는 **n개의 작업 $t_1, t_2, \cdots, t_n$**이 있고, **각 작업은 시작시간과 종료시간이** 있다.
- **작업 스케줄링 (Task Scheduling) 문제는 작업의 수행 시간이 중복되지 않도록 모든 작업을 가장 적은 수의 기계에 배정하는 문제이다.**

- 작업 스케줄링 문제는 학술대회에서 발표자들을 강의실에 배정하는 문제와 같다.
- **발표= '작업', 강의실= '기계'**

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- 작업 스케줄링 문제에 주어진 문제 요소
  - **작업의 수**
  - **각 작업의 시작시간과 종료시간**
  - **작업의 시작시간과 종료시간은 정해져 있으므로 작업의 길이도 주어진 것이다.**
- 여기서 작업의 수는 입력의 크기이므로 알고리즘을 고안하기 위해 고려되어야 하는 직접적인 요소는 아니다.
- 그렇다면, **시작시간, 종료시간, 작업 길이에** 대해 다음과 같은 그리디 알고리즘들을 생각해볼 수 있다.

- **빠른 시작시간 작업 우선 (Earliest start time first) 배정**
- **빠른 종료시간 작업 우선 (Earliest finish time first) 배정**
- **짧은 작업 우선 (Shortest job first) 배정**
- **긴 작업 우선 (Longest job first) 배정**

- 위의 4가지 중 첫 번째 알고리즘을 제외하고 나머지 3가지는 항상 최적해를 찾지 못한다.

작업 배정 알고리즘

JobScheduling

입력: n개의 작업 $t_1, t_2, \cdots, t_n$

출력: 각 기계에 배정된 작업 순서

1. **시작시간의 오름차순으로 정렬한 작업 리스트: L**
2. while (**L ≠∅**) {
3. **L에서 가장 이른 시작시간 작업 t_i를 가져온다.**
4. if (**t_i를 수행할 기계가 있으면**)
5. **t_i를 수행할 수 있는 기계에 배정한다.**
6. else
7. **새로운 기계에 t_i를 배정한다.**
8. **t_i를 L에서 제거한다.**

   }
9. return 각 기계에 배정된 작업 순서

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

- Line 1: 시작시간에 대해 **작업을 오름차순으로 정렬**
- Line 2~8의 while-루프는 **L에 있는 작업이 다 배정될 때까지 수행**된다.
- Line 3: **L에서 가장 이른 시작시간을 가진 작업 t_i를 선택**
- Line 4~5: **작업 t_i를 수행 시간이 중복되지 않게 수행할 기계를 찾아서, 그러한 기계가 있으면 t_i를 그 기계에 배정한다.**
- Line 6~7: **기존의 기계들에 t_i를 배정할 수 없는 경우에는 새로운 기계에 t_i를 배정한다**.
- Line 8: **작업 t_i를 L에서 제거하여, 더 이상 t_i가 작업 배정에 고려되지 않도록 한다.**
- Line 9: 마지막으로 각 기계에 배정된 작업 순서를 리턴

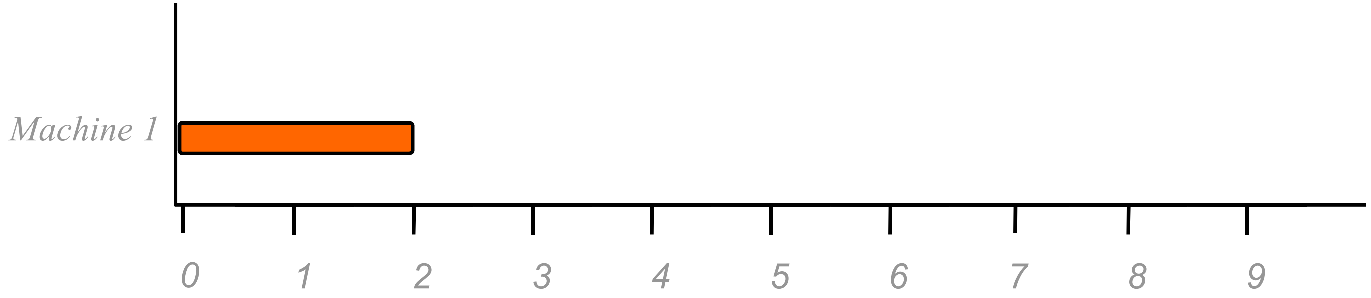
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

JobScheduling 알고리즘의 수행 과정

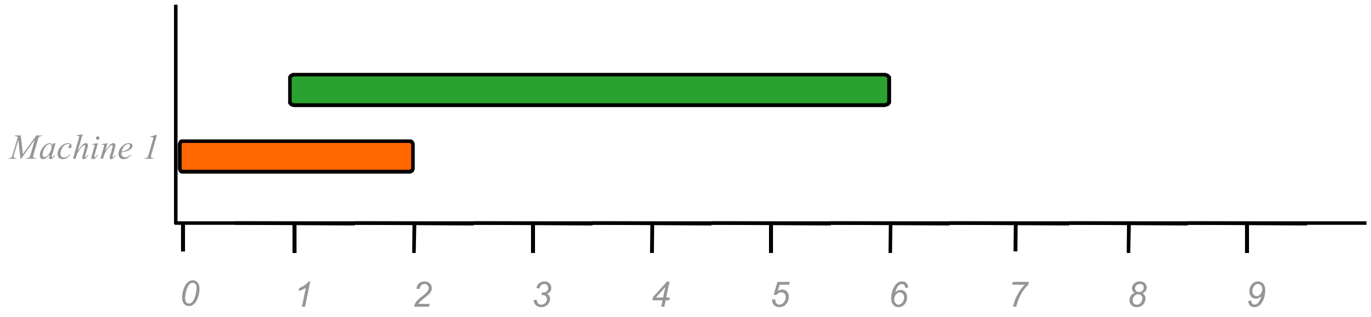
- **t_1=[7,8], t_2=[3,7], t_3=[1,5], t_4=[5,9], t_5=[0,2], t_6=[6,8], t_7=[1,6],**
- 단, **[s,f]**에서, **s는 작업의 시작시간이고, f는 작업 종료시간**

- Line 1: 시작시간의 오름차순으로 정렬한다. 따라서 **L = {[0,2], [1,6], [1,5], [3,7], [5,9], [6,8], [7,8]}**이다.
- 다음은 line 2~8까지의 while-루프가 수행되면서,
- **각 작업이 적절한 기계에 배정되는 것을 차례로** 보이고 있다.

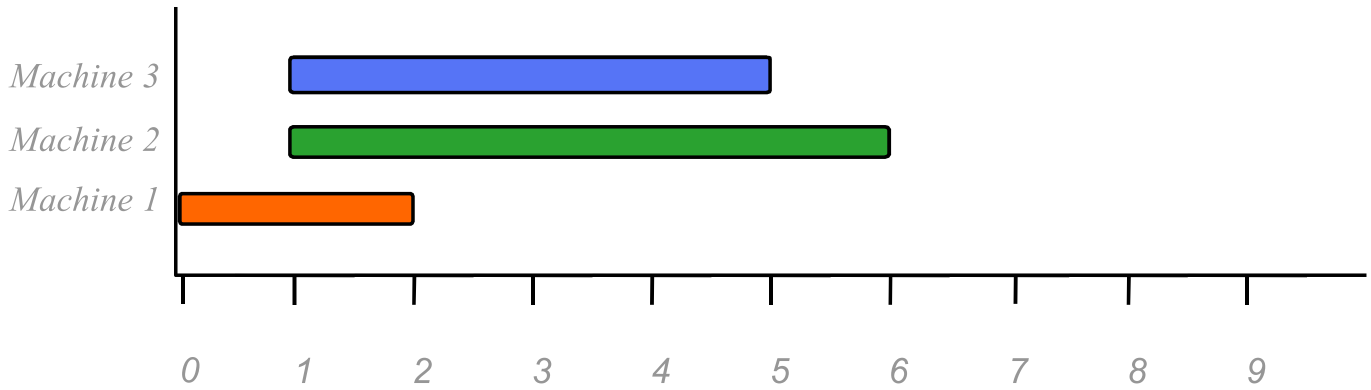
[0,2]



[0,2], [1,6]

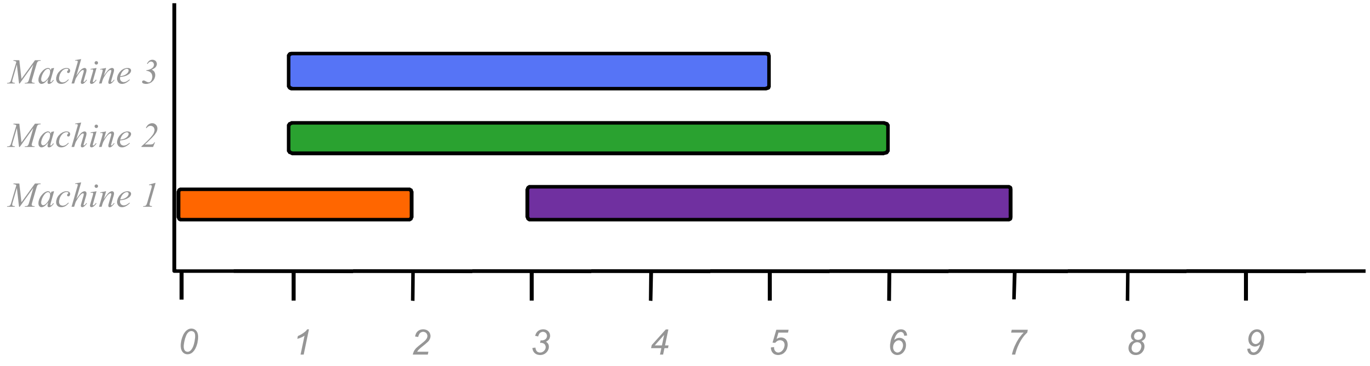


[0,2], [1,6], [1,5]

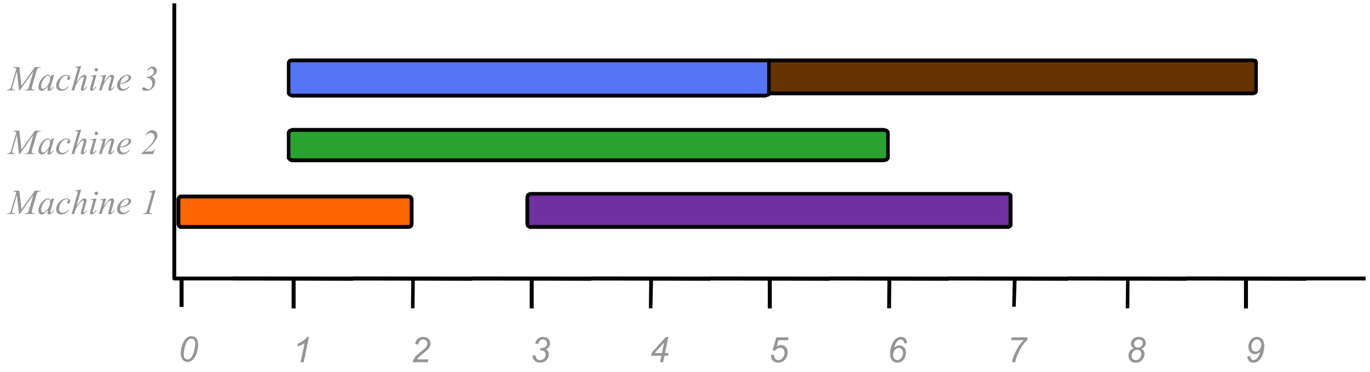


Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

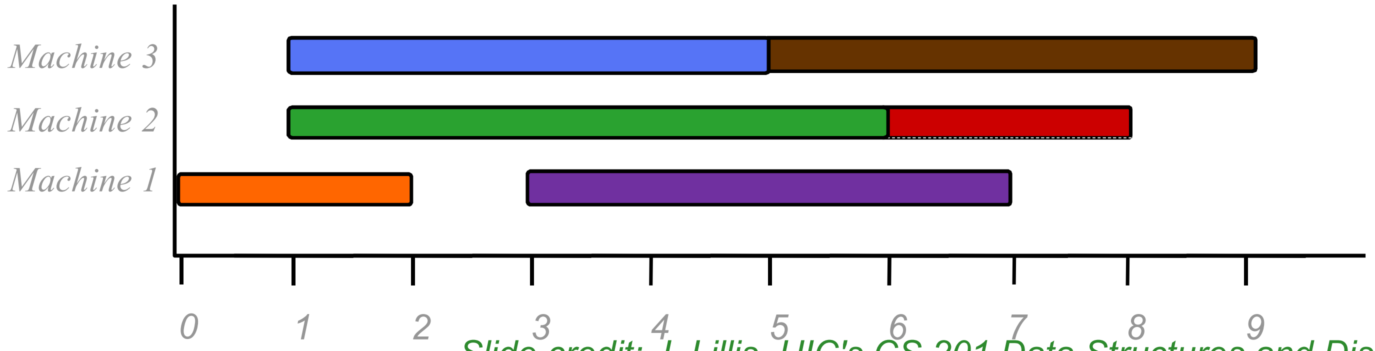
[0,2], [1,6], [1,5], [3,7]



[0,2], [1,6], [1,5], [3,7], [5,9]

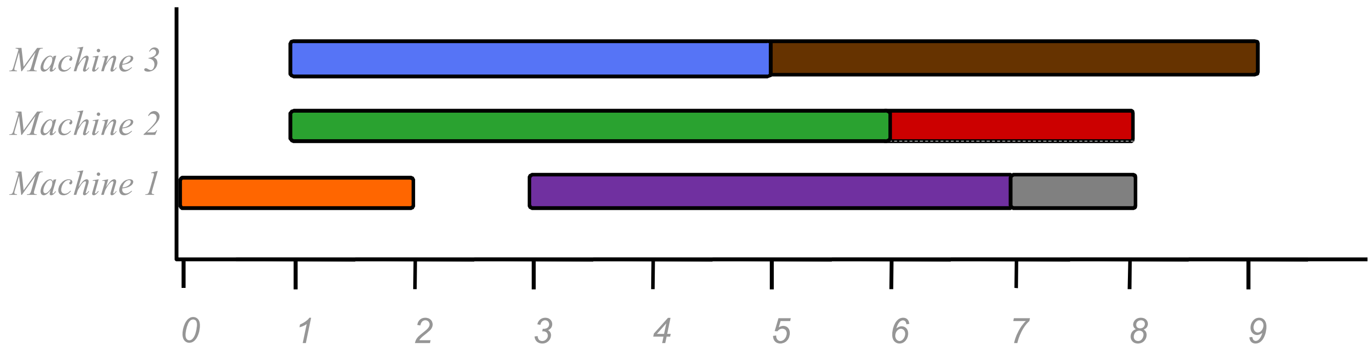


[0,2], [1,6], [1,5], [3,7], [5,9], [6,8]



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

[0,2], [1,6], [1,5], [3,7], [5,9], 6,8], [7,8]



시간복잡도

- Line 1에서 n개의 작업을 **정렬하는데 O(nlogn)** 시간이 걸리고,
- while-루프에서는 작업을 L에서 가져다가 수행 **가능한 기계를 찾아서 배정하므로 O(m)** 시간이 걸린다. 단, m은 사용된 기계의 수이다.
- while-루프가 수행된 총 횟수는 n번이므로, line 2~9까지는 **O(m)xn = O(mn) 시간이** 걸린다.
- 따라서 **JobScheduling 알고리즘의 시간복잡도는 O(nlogn)+O(mn)**이다.

응 용

- 비즈니스 프로세싱
- 공장 생산 공정
- 강의실/세미나 룸 배정
- 컴퓨터 태스크 스케줄링 등

요 약

- 그리디 알고리즘은 (입력) 데이터 간의 관계를 고려하지 않고 수행 과정에서 '욕심내어' 최적값을 가진 데이터를 선택하며, 선택한 값들을 모아서 문제의 최적해를 찾는다.
- 그리디 알고리즘은 문제의 최적해 속에 부분 문제의 최적해가 포함되어 있고, 부분 문제의 해 속에 그 보다 작은 부분 문제의 해가 포함되어 있다. 이를 최적 부분 구조 (Optimal Substructure) 또는 최적성 원칙 (Principle of Optimality)이라고 한다.
- 동전 거스름돈 문제를 해결하는 가장 간단한 방법은 남은 액수를 초과하지 않는 조건하에 가장 큰 액면의 동전을 취하는 것이다. 단, 일반적인 경우에는 최적해를 찾으나 항상 최적해를 찾지는 못한다.

- 부분 배낭 (Fractional Knapsack) 문제에서는 단위 무게 당 가장 값나가는 물건을 계속해서 배낭에 담는다. 마지막엔 배낭에 넣을 수 있을 만큼만 물건을 부분적으로 배낭에 담는다. 시간복잡도는 O(nlogn)이다.
- 집합 커버 (Set Cover) 문제는 근사 (Approximation) 알고리즘을 이용하여 근사해를 찾는 것이 보다 실질적이다. U의 원소들을 가장 많이 포함하고 있는 집합을 항상 F에서 선택한다. 시간복잡도는 $O(n^3)$이다.
- 작업 스케줄링 (Job Scheduling) 문제는 빠른 시작시간 작업 먼저 (Earliest start time first) 배정하는 그리디 알고리즘으로 최적해를 찾는다. 시간복잡도는 O(nlogn)+O(mn)이다. n은 작업의 수이고, m은 기계의 수이다.

END OF LECTURE 12

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I