KNU KYUNGPOOK NATIONAL UNIVERSITY

Lecture #17: Greedy Algorithm

School of Computer Science and Engineering
Kyungpook National University (KNU)

Woo-Jeoung Nam

탐욕법 (Greedy algorithm)

- 탐욕법: 단계마다 국부적으로 최적인 선택**(locally optimal choice)**을 하여, 문제 해결을 수행하는 경험론적 알고리즘 **(heuristic algorithm)**
  - ➢ 각 단계마다 지금 '당장' 가장 좋은 방법으로 보이는 것 만을 선택함 ↳ 대충은 적을 수 있음.
  - ➢ 분할정복 / 다이나믹 프로그래밍: 문제를 전체적인 입장에서 모든 경우를 고려함

선택 절차 → 타당성 조사 → 해법 조사.

- 탐욕법의 기본적인 접근법
  - ➢ **(1)** 선택 절차 **(Selection procedure)**
    - 특정 후보들 중에 하나를 선택하여, 해당 방식이 주어진 문제의 해인지 결정하는 절차
    - 해를 결정하는 방법: 현재의 상황에서 가장 최선의 경우 (앞/뒤를 고려하지 않음)
  - ➢ **(2)** 타당성 조사 **(Feasibility check)**
    - 선택한 해가 과연 타당한 것인가를 검사함 → 반례 찾기.
  - ➢ **(3)** 해법 조사 **(Solution Check)**
    - 지금까지 선택해왔던 해들이 문제에서 요구하는 조건에 만족하는 지를 검사함

탐욕법 (Greedy algorithm)

- **greedy algorithm**이 안되는 예시:
  - **Knapsack** (배낭 문제)
    - **0-1 Knapsack** 은 불가능
    - **Knapsack** 분할은 가능
- 그리디 알고리즘의 **3**가지 예시:
  - 행동 선택 **(Activity Selection)**
  - 잡 스케쥴링 **(Job Scheduling)**
  - 허프만 코딩 **(Huffman Coding)**

Knapsack - fractional

- 배낭에 담을 수 있을 만큼 물품들을 넣었을 때 **benefit(**가치**)**가 최대가 되는 짐을 고르는 것
- **benefit(**가치**) / weight(**무게**)** 의 비율을 각 물품마다 구한다
  - ➢ 비율이 높은 순으로 정렬을 한다
  - ➢ 배낭에 물품들을 비율 높은 순으로 넣는다.

10kg
10
20

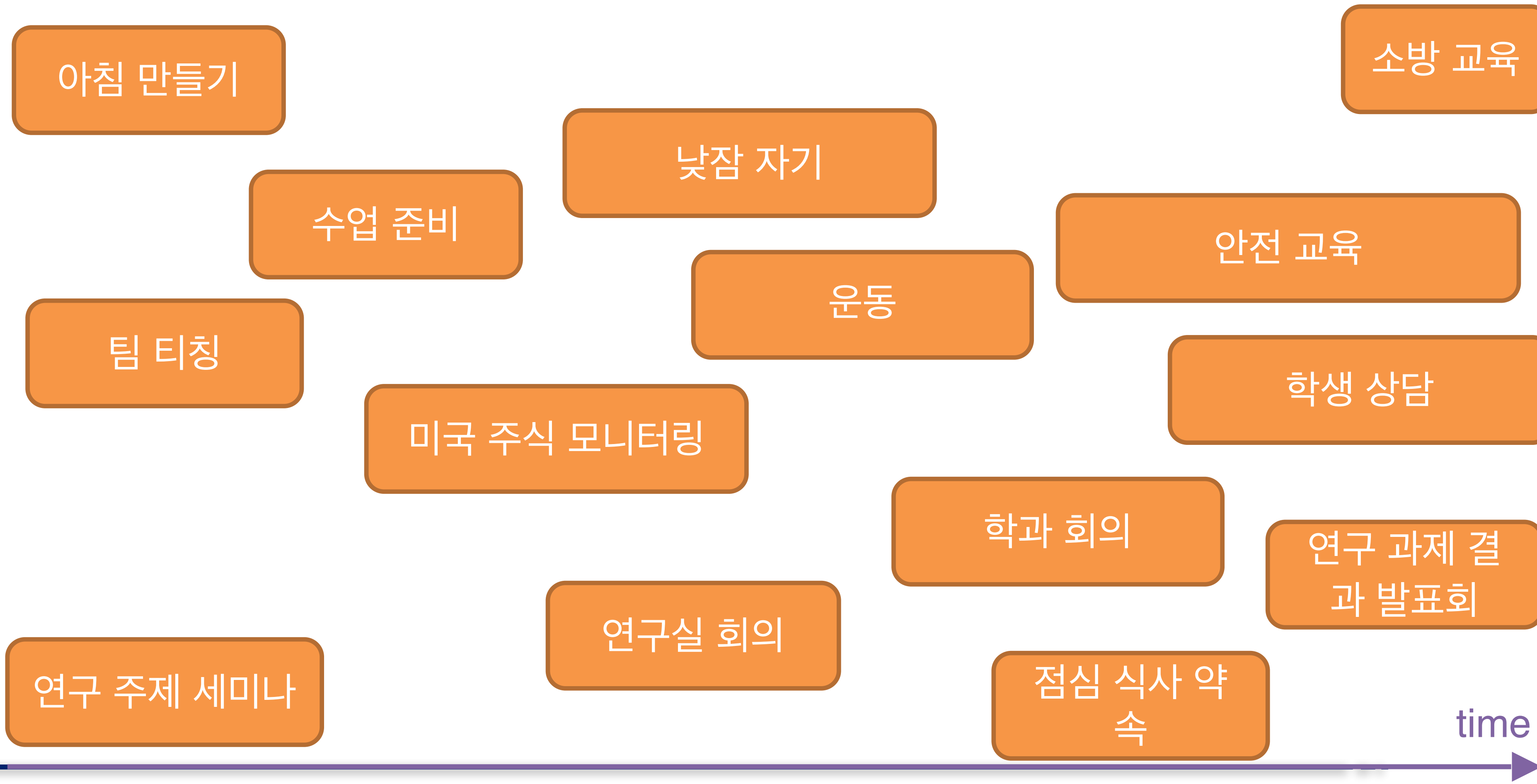
  - ➢ 만약에, 물품의 무게가 무게 최대치보다 크다면
    - **(**물품의 비율**) * (**최대치까지 남은 무게**)**를 구해서 답에 더해준다
  - ➢ 아니라면,
    - 물품을 그대로 배낭에 넣고, 배낭에 넣을 수 있는 최대치 무게 값 들어간 무게만큼 줄인다.

Knapsack - fractional

- 무게 당 이익이 높은 보석을 기준으로 먼저 넣으면 끝
- 배낭에 담을 수 있는 총 무게는 **30KG**
  - 아이템 **1 : 50**만 원**, 5KG**
  - 아이템 **2 : 60**만 원**, 10KG**
  - 아이템 **3 : 140**만 원**, 20KG**
- 키로당 가격
  - 아이템 **1 : 50/5 = 10**만 원
  - 아이템 **2 : 60/10 : 6**만 원
  - 아이템 **3 : 140/20 : 7**만 원
- 아이템 **1,** 아이템 **3**을 넣고**,** 아이템 **2**를 **5KG**만 넣으면 최적의 해답

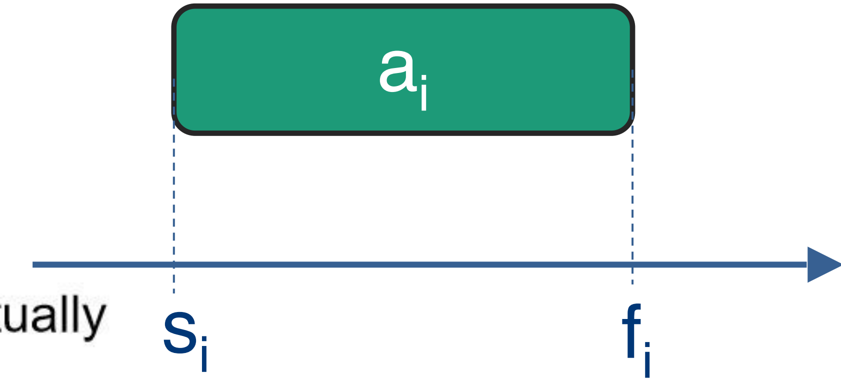
활동 선택 문제 (Activity Selection Problem)

- 활동 선택 문제란, 예를 들어, 한번에 하나의 활동만 처리할 수 있는 하나의 강의실에서 제안된 활동들 중 가장 많은 활동을 처리할 수 있는 스케줄을 짜는 문제



활동 선택 문제 (Activity Selection Problem)

- 특정 활동 a_i가 있으며, s_i는 활동의 시작시간, f_i는 활동이 끝나는 시간
- A_i, a_j 활동이 각각 존재할 때,
  두 활동의 활동시간이 서로 겹치면 안됩니다



- Problem Definition
  - Find the maximum-size subset of mutually compatible activities
- Activities a_i
  - s_i : start time
  - f_i : finish time
- a_i and a_j are mutually compatible
  → Intervals $[s_i,f_i)$ and $[s_j,f_j)$ do not overlap.
- Activities are stored in increasing order of f_i.

$$f_0 \le f_1 \le f_2 \le \cdots \le f_n \le f_{n+1}$$

활동 선택 문제 (Activity Selection Problem)

- **Input:**
  - ➢ Activities: $a_1, a_2, \ldots, a_n$
  - ➢ Start times: $s_1, s_2, \ldots, s_n$
  - ➢ Finish times: $f_1, f_2, \ldots, f_n$

- **Output:**
  - ➢ 그 날 가장 많은 활동을 할 수 있도록 선택한 조합

- **Shortest job first?**
- **Fewest conflicts first?**
- **Earliest start time first?**
- **Earliest ending time first?**

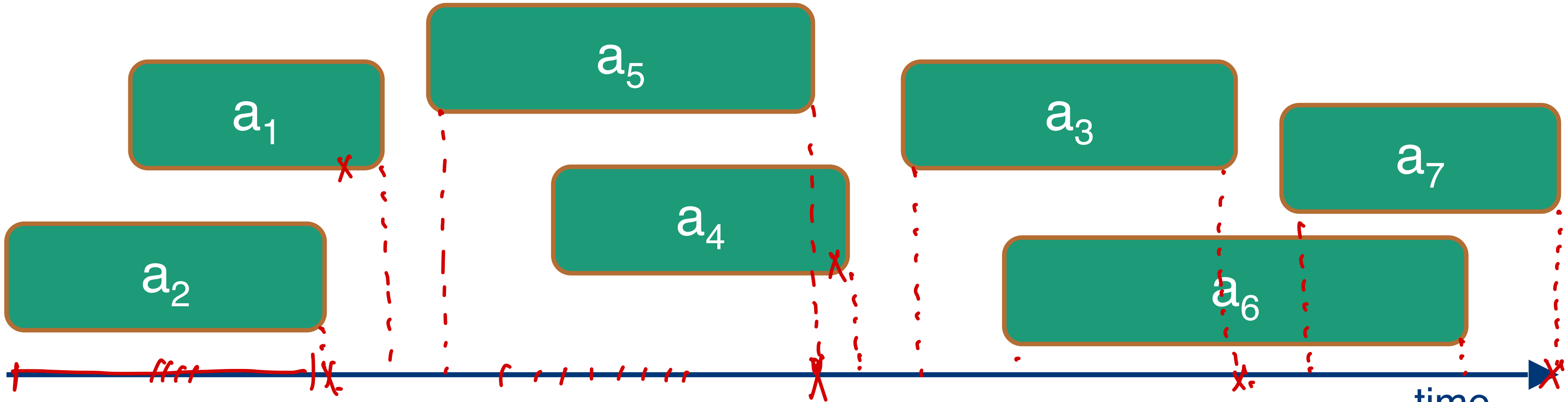
① Shortest job first:
② Fewest conflicts first
③ Earlist start time first
④ Earlist ending time first

활동 선택 문제 (Activity Selection Problem)

- 주어진 활동들을 종료 시간에 따라 오름차순으로 정렬.
  - 이렇게 하면 가장 먼저 끝나는 활동부터 고려 가능
- 가장 먼저 끝나는 활동을 선택
  - 이 활동은 선택된 활동들 중에서 겹치지 않으면서 가장 빨리 종료되므로, 다른 활동들을 선택할 수 있는 여유가 최대가 됩니다.
- 이전에 선택한 활동과 겹치지 않는 활동 중 가장 먼저 끝나는 활동을 찾아 선택
  - 이 과정을 활동들을 모두 검토할 때까지 반복
- 선택된 활동들의 목록을 반환
  - 이 목록은 최대한 많은 활동들을 포함하며, 겹치지 않는 조건을 만족합니다.

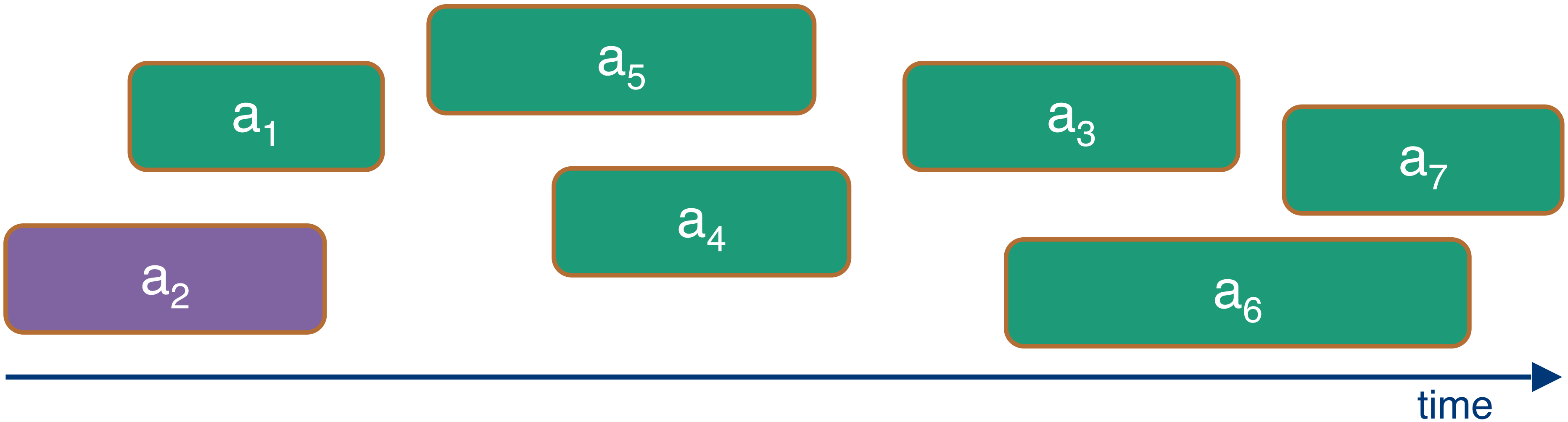
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



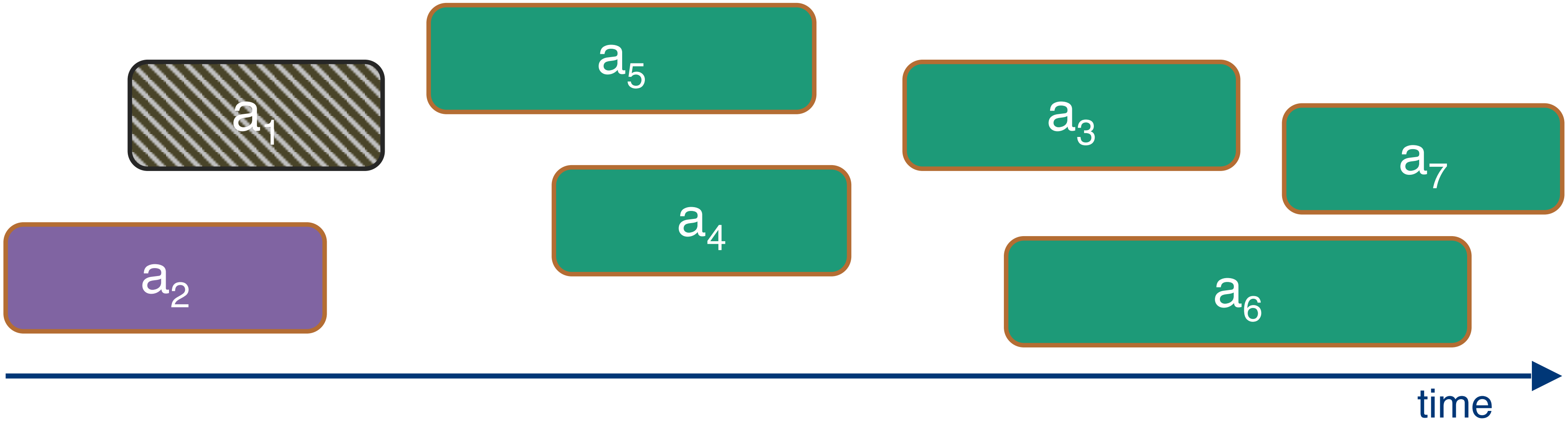
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



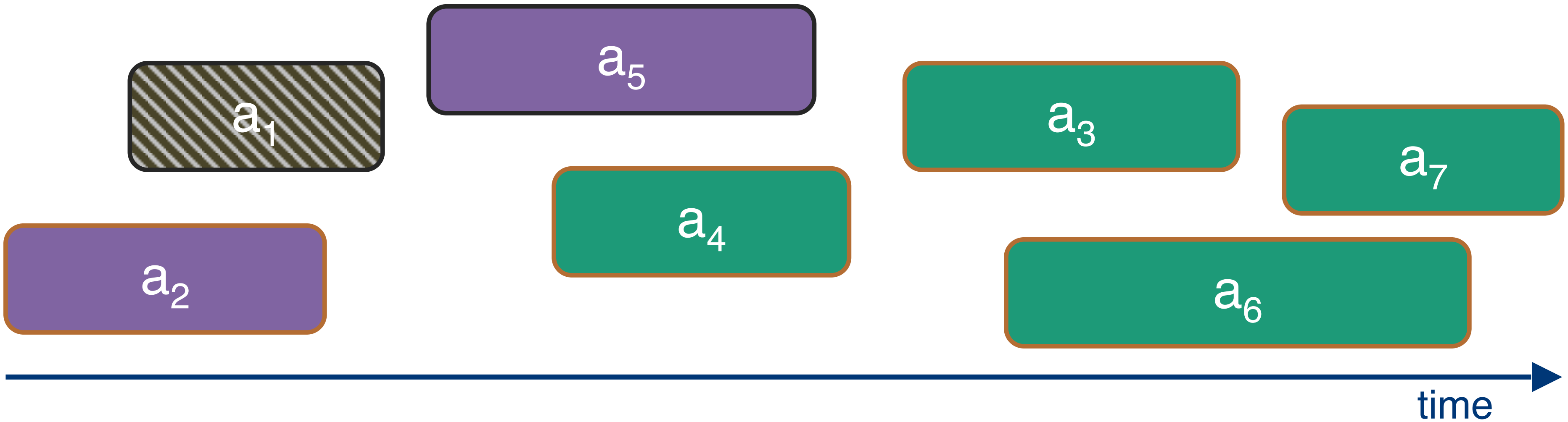
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



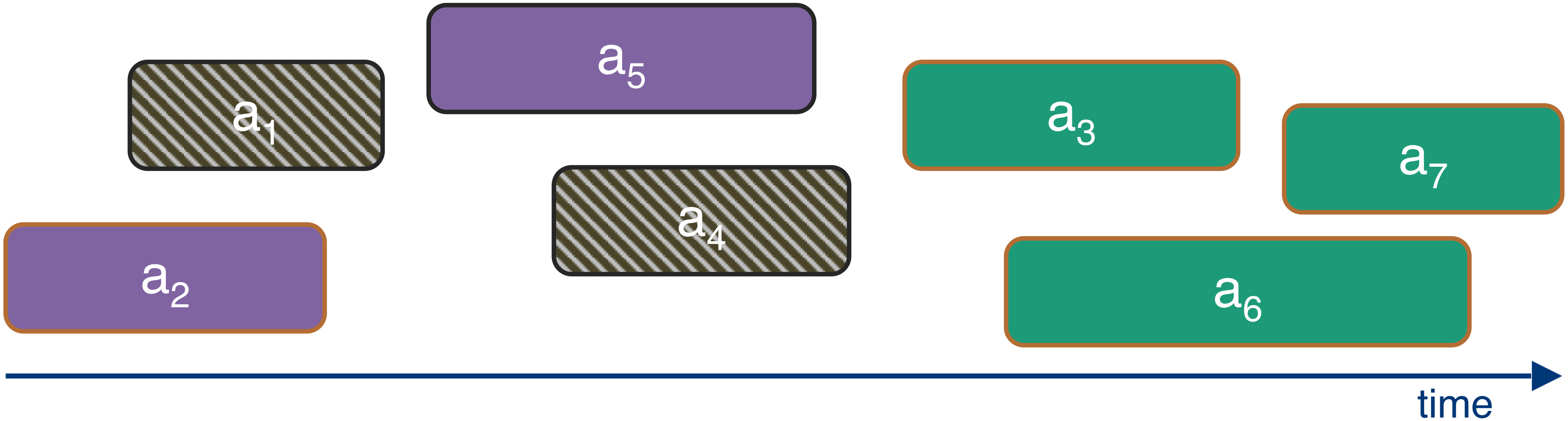
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - 해당 과정을 계속 반복



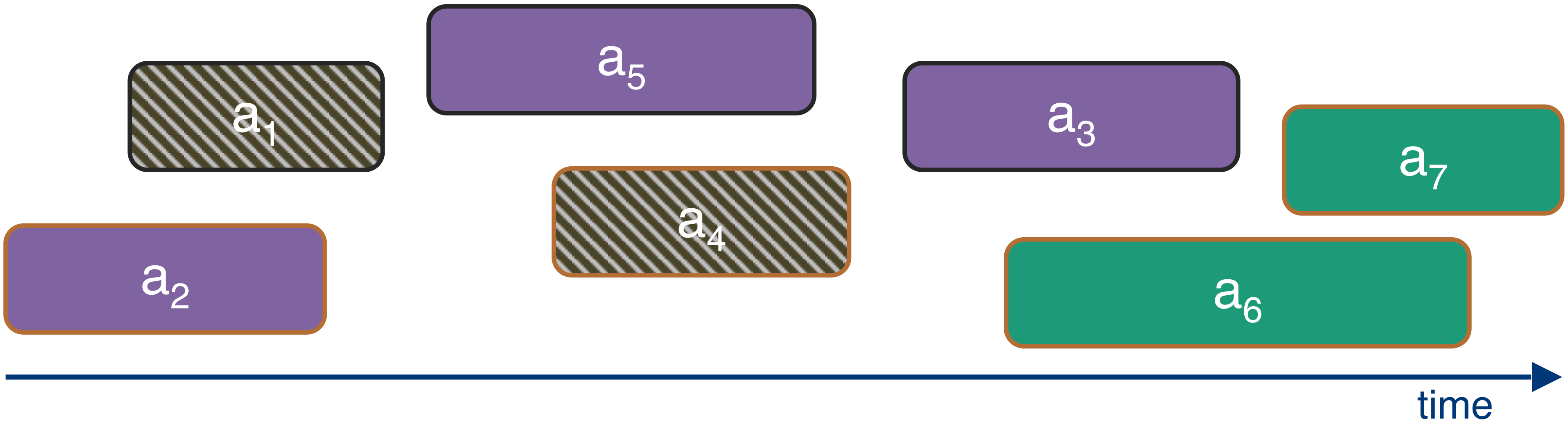
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



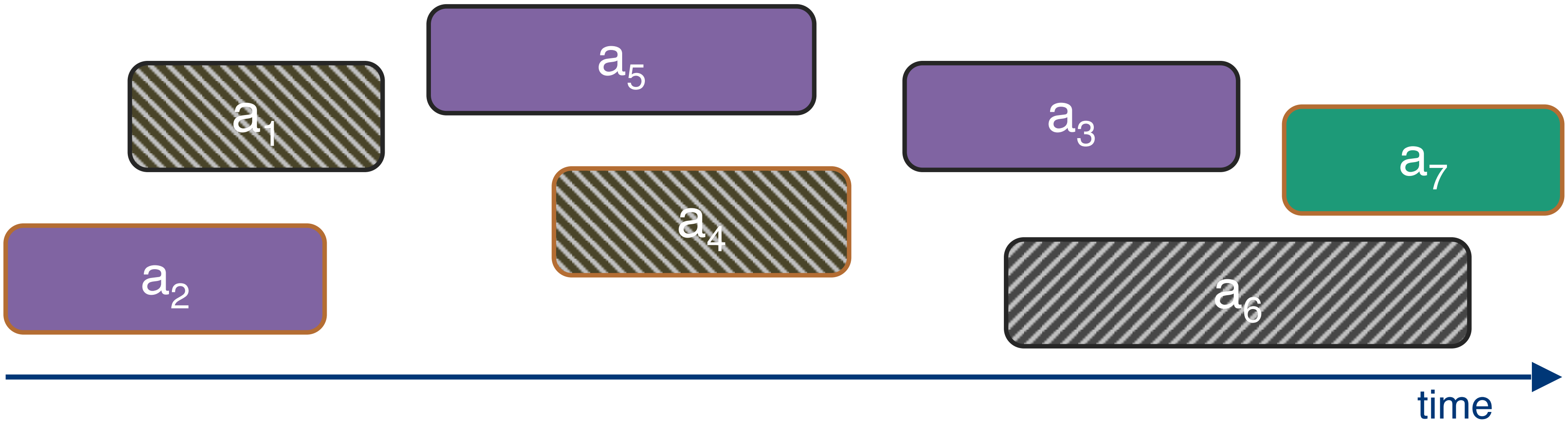
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



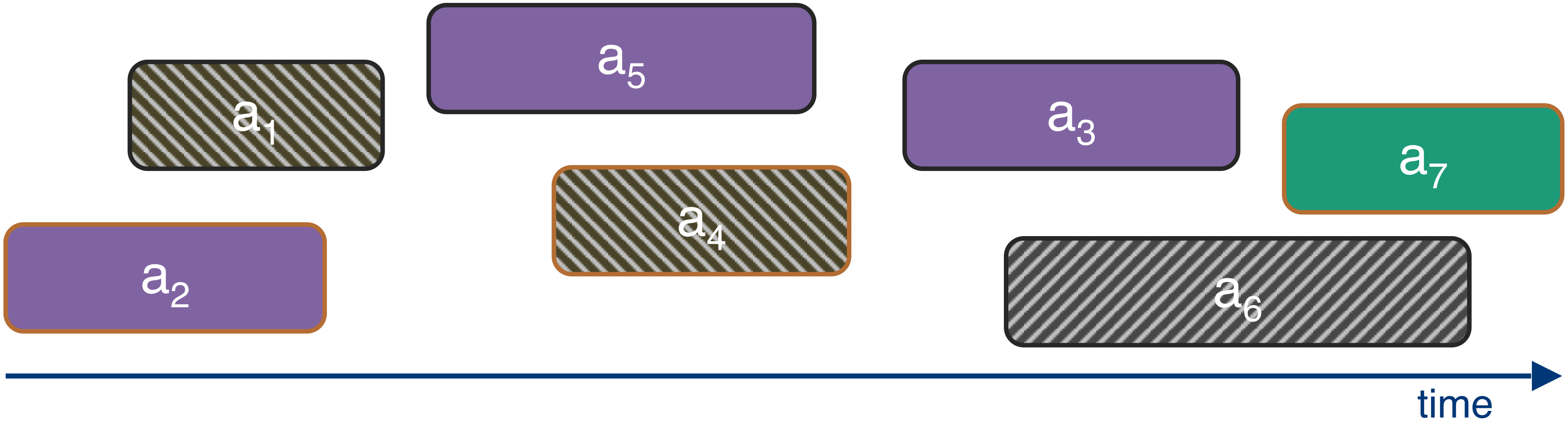
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



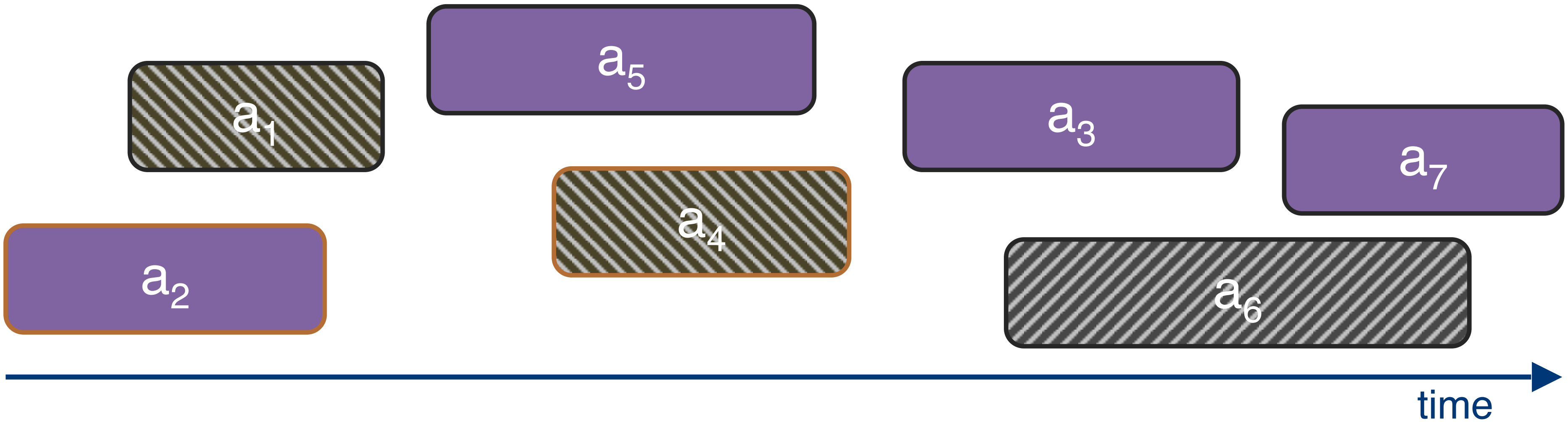
활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ (남은 활동들 중) 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ **(**남은 활동들 중**)** 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복



활동 선택 문제 (Activity Selection Problem)

- **Greedy Choice Property**
  - 각 단계에서 최적의 선택을 하면, 전역적인**(Global)** 최적해에 도달할 수 있다.

- **Optimal Substructure**
  - 문제 전체에 대한 최적해가 부분 문제에 대해서도 최적해여야 한다.

- 즉, 결정 상황에서 그리디 알고리즘을 통해 선택하는 방법 자체가 곧 문제 전체를 해결하는 방법과 같아야 한다.

- 시간복잡도
  - 일반적으로 **O(n log n)**
  - 정렬된 활동들에 대한 탐색은 **O(n)**
  - 따라서 전체 시간 복잡도는 **O(n log n)**

활동 선택 문제 (Activity Selection Problem)

- **Greedy algorithm**
  - ➢ **(**남은 활동들 중**)** 가장 빨리 끝나는 활동 들을 선택하기
  - ➢ 해당 과정을 계속 반복
- **a3, a9, a11**은 함께 스케줄로 짤 수 있는, 즉 양립할 수 있는 활동들
- 하지만 **a1, a4, a8, a11** 이라는 집합도 양립 가능하며 이러한 집합이 크기가 더 크기 때문에 최대 크기의 부분집합은 **a1, a4, a8, a11**

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

활동 선택 문제 (Activity Selection Problem)

■ 활동 a_i가 종료한 후에 시작하고, 활동 a_j가 시작하기 전에 종료하는 활동들의 집합을 S_{ij}

S_{ij}

a_i a_j

■ S_{ij}에 들어 있는 상호 양립 가능한 최대 크기의 집합을 찾기를 원하며, 그러한 최대 크기의 집합은 활동 a_k를 포함하는 A_{ij}라고 가정

a_k를 포함하는 A_{ij}가 해라 가정

- Subspace

$f_i \le s_k < f_k \le s_j$

식을 만족하는 a_k

$$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$$

A_{ik} to S_{ik}

A_{kj} to S_{kj}

A_{ij}

- Optimal substructure

Suppose an optimal solution A_{ij} to S_{ij} includes activity a_k. Then the solutions A_{ik} to S_{ik} and A_{kj} to S_{kj} must be optimal.

최적해 A_{ij}에 a_k를 포함하면

- Max-size subset

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

S_{ik}, S_{kj}가 남게됨

A_{ik} A_{kj}

$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

활동 선택 문제 (Activity Selection Problem)

- **Aik = Aij ∩ Sik**이고 **Akj = Aij ∩ Skj**
- **Aik**는 **Aij**에서 **ak**가 시작하기 전에 끝나는 활동들을 포함
- **Akj**는 **Aij**에서 **ak**가 끝난 후에 시작하는 활동들을 포함

$C[i,j] = C[i,k] + C[k,j]$

$C[i,j] = \max_{i \le k \le j} \{ C[i,k] + C[k,j] + 1 \}$

A_{ik}

재귀방정식 dp식은 이렇게 됨!

= 모든 부분 문제를 다 봐야 한다는 점이 문제임

- 만약 **DP**로 풀면?
  - ➢ 하지만 모든 부분문제를 풀어봐야한다

S_{ij}를 위한 최적 해의 크기

- Recursive Approach

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max_{i \le k \le j} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \ne \varnothing \end{cases}$$

부분 문제 중에 가장 큰 값을 취해야 한다.

$$C[i,j] = \begin{cases} 0 & S_{ij} = \phi \\ \max_{i \le k \le j} C[i,k] + C[k,j] + 1 & S_{ij} \ne 0 \end{cases}$$

Looks like a DP problem

모든 부분 문제

활동 선택 문제 (Activity Selection Problem)

- 동적 프로그래밍에서 필요하지 않은 케이스 조차 탐색하는 불필요한 탐색시간을 없애고, 각 단계에서 최적의 선택을 함으로써 시간적으로 효율적

"c[k, n]" → 2개 수 있는 m 을 찾음

한개

c[k, m] + c[m, n]

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]    // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

s[m] ≥ f[k] 인 것 부터 시작

return {am} ∪ recursive(s, f, m, n.)

O(n)

그리디 알고리즘

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

k=1
m=2 ~ n
s[m] ≥ f[k]
A = A ∪ {am}
k=m

A = {a1}
k=1
m=2 to n.
if. s[m]≥f[k]
A = A∪{am}
k=m.
return A

→더 쉽다.

k=1
for m=2 ~n
if. s[m]≥f[k]
A = A ∪ {am}
k=m

활동 선택 문제 (Activity Selection Problem)

- 가상의 **a0**가 끝났다고 가정, 첫번째 **Recursive-activity** 호출
- **a1**이 호출되고 선택된 활동은 회색으로 마킹
- 화살표 방향에 따라 버리거나 선택
  - ➢ (왼쪽은 버리고 오른쪽은 선택)