Lecture 19 - Greedy Algorithms (Activity Selection and Scheduling)

Fall 2025, Korea University

Instructor: Gabin An (gabin_an@korea.ac.kr)

Course Outline (After Midterm)

- Part 3: Data Structures
  - Graphs, Graph Search (DFS, BFS) and Applications (Finding SSCs w/ DFS)
- Part 4: Dynamic Programming
  - Shortest-Path: Dijkstra, Bellman-Ford, Floyd-Warshall Algorithms
  - More General DP: Longest Common Subsequence, Knapsack Problem
- Part 5: Greedy Algorithms and Others
  - **Activity Selection, Scheduling** 👈, Optimal Codes
  - Minimum Spanning Trees
  - Max Flow, Min Cut and Ford-Fulkerson Algorithms
  - Stable Matching, Gale-Shapley Algorithm

Agenda

- What is a **Greedy Algorithm**?
- New problems: **Activity Selection** and **Scheduling**

What is a Greedy Algorithm?

- A greedy algorithm is an approach for solving a problem by selecting **the best option available at each step**.
- It builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
- Greedy algorithms make locally optimal choices **with the hope of finding a global optimum**.

Example: Dijkstra's Algorithm for SSSP

Key Idea:

Maintain distance estimates $d[t]$ for all nodes $t \in V$.

- Initially
  - $d[s] \leftarrow 0$, and $d[t] \leftarrow \infty$ for all $t \neq s$
  - $F \leftarrow V$ (unfinalized nodes), $D \leftarrow \emptyset$ (finalized nodes)
- Main Loop (while $F \neq \emptyset$):
  i. Select $x \in F$ with the smallest $d[x]$ (⭐ **Chossing BEST OPTION available**)
  ii. Relax every edge (x, y): $d[y] \leftarrow \min(d[y], d[x] + w(x, y))$
  iii. Move x from F to D (finalized set)

Advantages and Drawbacks of Greedy Approach

- Advantages:
  - **Fast execution**: Only one choice is made at each step → no need to explore many alternatives → runs very quickly.
  - **Simplicity**: Easy to design, describe, and implement compared to more complex strategies (e.g., dynamic programming).
  - **Good for specific problems**: For problems with the greedy-choice property and optimal substructure, greedy always guarantees the optimal solution.
    - We need to prove correctness.
- Drawbacks:
  - **Not universally correct**: For many problems (e.g., 0-1 Knapsack), greedy may produce suboptimal results.

Property of Problems Suitable for Greedy Algorithms

- **Greedy Choice Property**
  - A problem can be solved using a greedy approach if we can make a "greedy", locally-optimal choice, and guarantee that a globally-optimal solution still exists.
- **Optimal Substructure**
  - If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach.

> Greedy algorithms can be seean as a refinement of dynaming programming.

To Give a Correct Greedy Algorithm

- One must first identify optimal substructure (as in dynamic programming)
- and then prove that at each step, **you only need to consider one subproblem**.

Even though there may be many possible subproblems to recurse on, **given our selection of subproblem**, there is always **a globally-optimal solution** that contains the **optimal solution to the selected subproblem**.

Problems with Nice (Correct) Greedy Algorithms

- **Activity Selection** 👈
- Scheduling 👈
- Optimal Codes
- Minimum Spanning Trees
- ... and many more

What's the Activity Selection Problem?

We have n **activities**, $a_1, a_2, \ldots, a_n$, where a_i has:

- **Start time** s_i and **Finish time** f_i

Goal:

- Select the **maximum number of non-conflicting activities**

Constraint:

- A person can do at most **one activity at a time**

Example: Movie Marathon 🎬

> You're at a film festival, and you want to watch as many movies as possible.

- Each movie has a start and end time. You must watch the **full movie**!!
- You can watch at most one movie at a time.
- The ticket was so expensive 💰, so you want to **maximize number of movies seen**.
- Movie Time Table (start, end):

```
a_1. Avatar: The Way of Water (11:00, 14:12)
a_2. La La Land               (11:50, 13:58)
a_3. Casablanca               (12:00, 13:42)
a_4. Dune: Part 2             (14:00, 16:46)
a_5. A Star is Born           (14:10, 16:25)
a_6. Sing Street              (16:40, 18:26)
```

First Attempt: Dynamic Programming

Input: n **activities**, $\{a_1, a_2, \ldots, a_n\}$, where a_i has start time s_i and finish time f_i

Define $A_{i,j}$ = the largest set of **non-conflicting** activities **between** activity a_i and a_j.

Recurrence:

$$|A_{i,j}| = \max_{k \in S_{i,j}} \{|A_{i,k}| + 1 + |A_{k,j}|\}$$

- where $S_{i,j}$ = activities that can fit between a_i and a_j
  - $S_{i,j} = \{a_k | f_i \leq s_k, f_k \leq s_j\}$
- Complexity: $O(n^3)$ (too slow!)
  - Need to fill in a $n \times n$ table
  - Filling a single entry might take $O(n)$ time

Greedy Insight

Instead of trying all activities, always pick the activity with the **earliest finishing time**.

Why?

- Leaves the most room for the remaining activities
- Always part of some optimal solution (proof soon!)

Algorithm (Greedy-Activity-Selection)

1. Sort activities by **finish time**
2. Pick the first activity
3. Repeatedly add the next activity that starts after the last one finishes

Let's Go Back to Our Movie Marathon Example 🎬

```
a_1. Avatar: The Way of Water (11:00, 14:12) ✅ Pick the movie that starts first
a_2. La La Land               (11:50, 13:58) ❌ overlaps
a_3. Casablanca               (12:00, 13:42) ❌ overlaps
a_4. Dune: Part 2             (14:00, 16:46) ❌ overlaps
a_5. A Star is Born           (14:10, 16:25) ❌ overlaps
a_6. Sing Street              (16:40, 18:26) ✅ 🥹
```

Activities sorted by finish time:

```
a_3. Casablanca               (12:00, 13:42) ✅ Pick the movie that finishes first
a_2. La La Land               (11:50, 13:58) ❌ overlaps
a_1. Avatar: The Way of Water (11:00, 14:12) ❌ overlaps
a_5. A Star is Born           (14:10, 16:25) ✅
a_4. Dune: Part 2             (14:00, 16:46) ❌ overlaps
a_6. Sing Street              (16:40, 18:26) ✅ 😆
```

Quick Quiz

Given the following activities (start, finish):

```
a1: (1, 4)
a2: (3, 7)
a3: (0, 2)
a4: (4, 5)
a5: (8, 9)
a6: (5, 7)
a7: (5, 8)
```

Q. Which activities will the greedy algorithm select?

Let's Prove the Correctness ...

- $A_{i,j}$ = the largest set of **non-conflicting** activities **between** activity a_i and a_j.
- $S_{i,j}$ = activities that can fit between a_i and a_j
  - $S_{i,j} = \{a_k | f_i \leq s_k, f_k \leq s_j\}$. Therefore, $A_{i,j} \subset S_{i,j}$

```
a1: (1, 4)
a2: (3, 7)
a3: (0, 2)
a4: (4, 5)
a5: (8, 9)
a6: (5, 7)
a7: (5, 8)
```

- $S_{3,5} = \{a_4, a_6, a_7\}$
- $A_{3,5} = \{a_4, a_6\}$

Proposition:

> **For each $S_{i,j}$, there is an optimal solution $A_{i,j}$ containing $a_k \in S_{i,j}$ of minimum finishing time f_k.**

> Example:
>
> - $S_{3,5} = \{a_4, a_6, a_7\}$ (a_4 has the earliest finishing time)
> - $A_{3,5} = \{a_4, a_6\}$

If this proposition is true, when f_k is minimum, then $A_{i,k}$ is empty because no activities can finish before a_k. Thus:

$$|A_{i,j}| = |A_{i,k}| + 1 + |A_{k,j}| = 0 + 1 + |A_{k,j}| = 1 + |A_{k,j}|$$

$$A_{i,j} = \{a_k\} \cup A_{k,j}$$

> For each $S_{i,j}$, there is an optimal solution $A_{i,j}$ containing $a_k \in S_{i,j}$ of minimum finishing time f_k.

Proof:

- Let a_k be the activity of minimum finishing time in $S_{i,j}$.
- Let $A'_{i,j}$ be **some** maximum set of non-conflicting activities where a_l is the activity of minimum finishing time in $A'_{i,j}$.
- Consider $A_{i,j} = A'_{i,j} \setminus \{a_l\} \cup \{a_k\}$. It is clear that $|A_{i,j}| = |A'_{i,j}|$ (*maximum*).

> We only need to prove $A_{i,j}$ does not have conflicing activities.

- We know $a_l \in A'_{i,j} \subset S_{i,j} \rightarrow f_l \geq f_k$, as a_k has the minimum finishing time in $S_{i,j}$.
- We know $\forall a_t \in A'_{i,j} \setminus \{a_l\}, s_t \geq f_l$. Therefore, $s_t \geq f_k$, and a_t does not conflict with a_k. Thus, $A_{i,j}$ is an **optimal** solution.

Algorithm (Greedy-Activity-Selection)

1. Sort activities by **finish time**
2. Pick the first activity
3. Repeatedly add the next activity that starts after the last one finishes

Time Complexity

- Sorting by finish time: $O(n \log n)$
- Scanning once: $O(n)$
- **Total:** $O(n \log n)$
    - Faster than the original DP solution.

Problems with Nice (Correct) Greedy Algorithms

- Activity Selection ✅
- **Scheduling** 👈
- Optimal Codes
- Minimum Spanning Trees
- ...

New Problem: Scheduling

We are given n **jobs**, where each job j has:

- **Length** l_j (time required) and **Weight** w_j (priority / importance)

Constraint:

- Jobs are executed one after another (1 machine)

Completion time of job j:

$$c_j = (\text{sum of lengths up to job j})$$

Goal: **Minimize weighted sum of completion times**

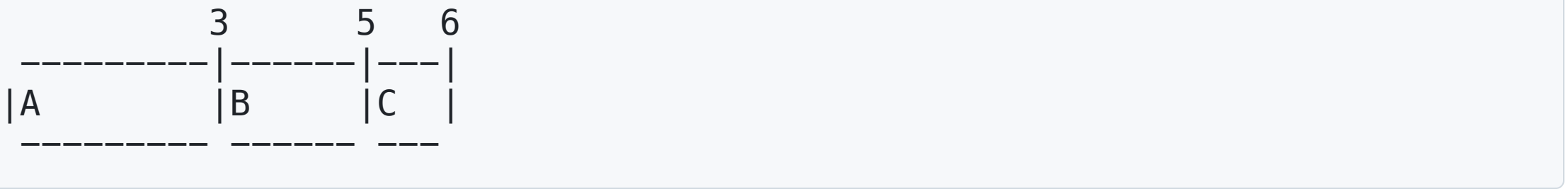
$$\min \sum_j w_j c_j$$

Why Does This Matter?

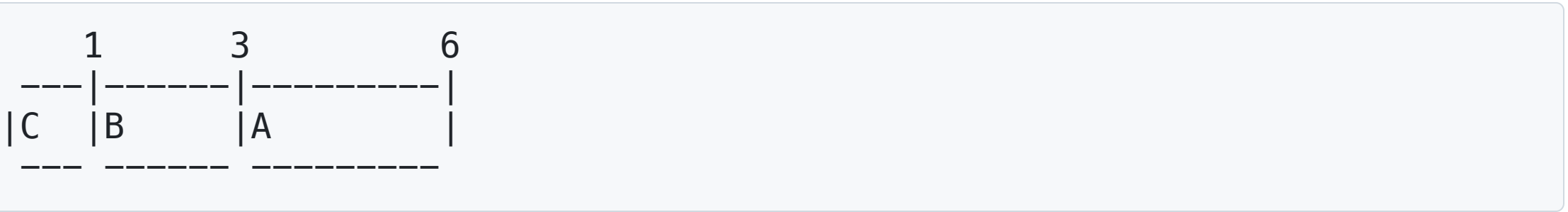
- CPU scheduling (OS)
- Printer queues
- Task prioritization in project management
- Packet transmission in networks
- Anytime we balance **urgency (weight) vs. duration (time)**

Example

Jobs: A ($l = 3, w = 4$), B ($l = 2, w = 2$), C ($l = 1, w = 5$)



- Weighted sum of completion time: $4 \times 3 + 2 \times 5 + 5 \times 6 = 12 + 10 + 30 = 52$



- Weighted sum of completion time: $4 \times 6 + 2 \times 3 + 5 \times 1 = 24 + 6 + 5 = 35$

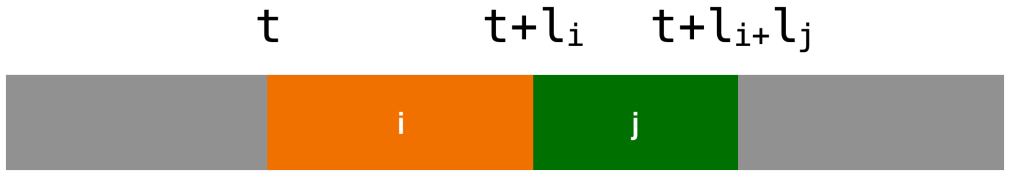
Intuition

Our goal is to minimize the weighted sum of completion times.

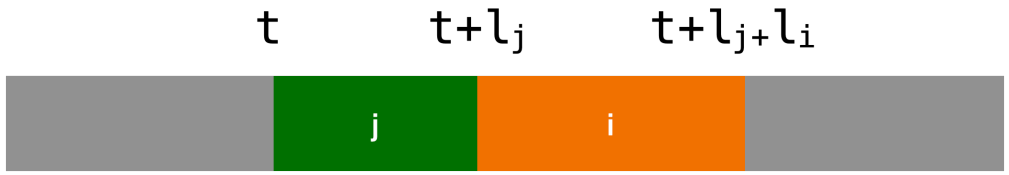
- If **all lengths equal** → put higher weights first
- If **all weights equal** → put shorter jobs first
- But when both differ…?
- Especially, what do we do in the cases where $l_i < l_j$ (i should be preferred) and $w_i < w_j$ (j should be preferred)? 🤔
  → **shorter job versus more important job**
  → Need a ratio that balances weight & length!

Obtimal Substructure - Swap Argument

Suppose we have a job i followed by job j in the **optimal** order, and job i is started at t.



Consider swapping jobs i and j:



- Job i's completion time: $t + l_i \rightarrow t + l_j + l_i$ ($\Delta = +l_j$)
- Job j's completion time: $t + l_i + l_j \rightarrow t + l_j$ ($\Delta = -l_i$)

> Swapping jobs i and j does not alter the completion times for every other job.

Obtimal Substructure - Swap Argument

- Job i's completion time: $t + l_i \rightarrow t + l_j + l_i$ ($\Delta = +l_j$)
- Job j's completion time: $t + l_i + l_j \rightarrow t + l_j$ ($\Delta = -l_i$)

Our objective function changes as follows:

$$\Delta\text{Objective} = w_i \cdot l_j + w_j \cdot (-l_i) = w_i l_j - w_j l_i.$$

Since the original order was optimal:

$$w_i l_j - w_j l_i \geq 0 \quad \Longrightarrow \quad \frac{l_j}{w_j} \geq \frac{l_i}{w_i}$$

Therefore, we want to process jobs in increasing order of $\frac{l}{w}$.

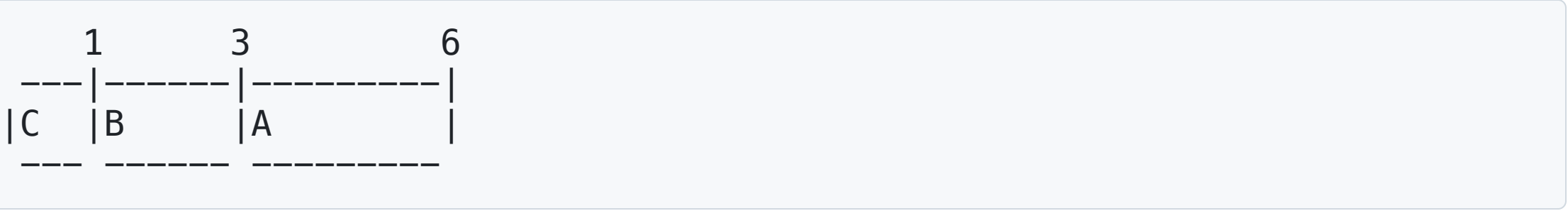
Greedy Scheduling Algorithm

1. Compute ratio l_j/w_j for each job
2. Sort jobs by this ratio (ascending)
3. Execute jobs in that order

Time Complexity

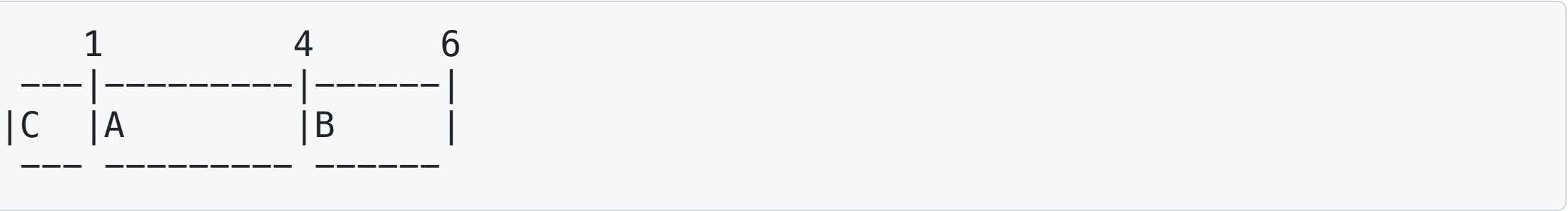
- Sorting: $O(n \log n)$
- Schedule evaluation: $O(n)$

Example: A ($l = 3, w = 4$), B ($l = 2, w = 2$), C ($l = 1, w = 5$)



- Weighted sum of completion time: $4 \times 6 + 2 \times 3 + 5 \times 1 = 24 + 6 + 5 = 35$

Ratios: A ($= \frac{3}{4}$) 2, B ($= 1$) 3, C ($= \frac{1}{5}$) 1



- Weighted sum of completion time: $4 \times 4 + 2 \times 6 + 5 \times 1 = 16 + 12 + 5 = 33$ 👑

Quick Quiz 🤓

Jobs:

- X: $l = 4, w = 4$
- Y: $l = 2, w = 3$
- Z: $l = 1, w = 10$

What is the optimal order that minimizes the total weighted sum of completion times?

Next Time

- Activity Selection ✅
- Scheduling ✅
- **Optimal Codes** 👈
- Minimum Spanning Trees
- ...

Credits & Resources

Lecture materials adapted from:

- Stanford CS161 slides and lecture notes
  - https://stanford-cs161.github.io/winter2025/
- *Algorithms Illuminated* by Tim Roughgarden
  - https://algorithmsilluminated.com/