Chapter 15. Greedy Algorithms

Joon Soo Yoo

May 22, 2025

Assignment

- Read §15.1
- Problems
  - §15.1 - 5

Chapter 15: Greedy Algorithms

- **Chapter 15.1: An Activity-Selection Problem**
- Chapter 15.2: Elements of the Greedy Strategy
- Chapter 15.3: Huffman Codes
- Chapter 15.4: Offline Caching

Chapter 15: Greedy Algorithms

$(A_1 \cdots A_n)$

$A_{1:k}$ $\quad A_{k+1:n}$

$\min_k \left(m[1][k] + m[k+1][n] + p_{i-1} p_k p_j \right)$

- Greedy algorithms solve optimization problems by making locally optimal choices.
- These choices are made without considering future consequences in detail.
- Often simpler and more efficient than dynamic programming.
- Greedy algorithms work for a wide range of problems:
  - Activity selection
  - Huffman coding
  - Offline caching
  - Minimum spanning tree, Dijkstra's algorithm
- Greedy choice doesn't always guarantee global optimality—but in some problems, it does.

$l = 2 \cdots n - l + 1$

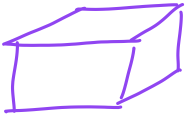
$m[i][j]$

$m[1][n]$

Greedy Algorithm vs Dynamic Programming

- **Dynamic Programming:**
  - Explores all subproblems and stores their results.
  - Bottom-up or memoized top-down.
- **Greedy Algorithm:**
  - Makes a single "best-looking" choice at each step.
  - Top-down: make a choice, then solve the remaining subproblem.
- This chapter shows when and why greedy algorithms yield optimal solutions.

15.1 An Activity-Selection Problem



- **Goal:** Select a maximum-size set of compatible activities.
- Each activity a_i has:
  - Start time s_i
  - Finish time f_i
  - Interval $[s_i, f_i)$

- Activities a_i and a_j are **compatible** if:

$$s_i \geq f_j \quad \text{or} \quad s_j \geq f_i$$

- Only one activity can use the resource at a time.

Activity-Selection Problem Example

$a_1 \quad a_2 \qquad\qquad a_n$

$s_1 \quad s_2 \quad \dots \quad s_n$

$f_1 \le f_2 \le \quad \le f_n$

- Suppose we are given a set of n activities:

$$S = \{a_1, a_2, \ldots, a_n\}$$

- Assume activities are sorted by finish time:

$$f_1 \le f_2 \le \cdots \le f_n \tag{15.1}$$

- Goal: Select the largest subset of mutually compatible activities.

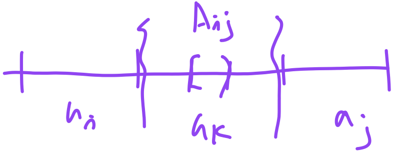
Figure: Activity Data Table

a_1 a_4 a_9 a_{11}

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- Each a_i represents an activity with a start and finish time.
- The table is sorted by increasing finish time.
- Example compatible subsets:
  - $\{a_3, a_9, a_{11}\}$ — not maximum
  - $\{a_1, a_4, a_8, a_{11}\}$ — maximum
  - $\{a_2, a_4, a_9, a_{11}\}$ — maximum

Optimal Substructure of the Activity-Selection Problem

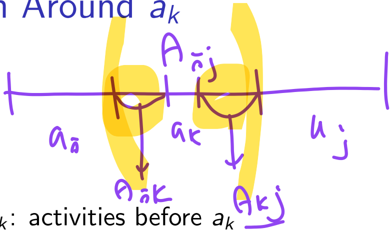


- Let S_{ij} be the set of activities that:
  - Start after a_i finishes
  - Finish before a_j starts

$\bar{A}_{ij}$

- Let A_{ij} be a maximum set of mutually compatible activities in S_{ij}.
- Suppose A_{ij} includes some activity a_k.

Dividing the Problem Around a_k



- If $a_k \in A_{ij}$, then:
  - $A_{ik} = A_{ij} \cap S_{ik}$: activities before a_k
  - $A_{kj} = A_{ij} \cap S_{kj}$: activities after a_k
- Then:

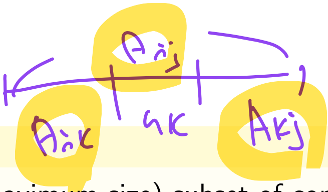
$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

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

- So the size of the optimal solution:

$$|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$$

Claim: Optimal Substructure in Activity Selection



Claim

If A_{ij} is an optimal (maximum-size) subset of compatible activities in S_{ij} and includes some activity a_k, then the subsets:

- $A_{ik} \subseteq S_{ik}$ (activities before a_k)
- $A_{kj} \subseteq S_{kj}$ (activities after a_k)

must also be optimal for their respective subproblems.

Proof: Cut-and-Paste Argument

- Assume $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ is optimal
- Suppose, for contradiction, a better solution A'_{kj} exists:

$$|A'_{kj}| > |A_{kj}|$$

- Construct a new solution:

$$A'_{ij} = A_{ik} \cup \{a_k\} \cup A'_{kj}$$

- Then:

$$|A'_{ij}| = |A_{ik}| + 1 + |A'_{kj}| > |A_{ik}| + 1 + |A_{kj}| = |A_{ij}|$$

- Contradiction! So A'_{kj} cannot exist.
- Therefore, A_{kj} must be optimal. Same logic applies to A_{ik}.

Conclusion: Optimal Substructure Holds

- From the cut-and-paste argument, we have shown:

Optimal Substructure ← DP ✓

If A_{ij} is optimal, then:

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

where A_{ik} and A_{kj} are also optimal.

- This property enables both:
  - A recursive (or DP) solution, and
  - A greedy solution, which we explore next.

Dynamic Programming Recurrence

$|A_{ij}| = c[i,j]$

- Let $c[i,j]$ be the size of an optimal solution for S_{ij}.
- If we know a_k is in the solution:

$$\underbrace{c[i,j]}_{|A_{ij}|} = \underbrace{c[i,k]}_{|A_{ik}|} + \underbrace{c[k,j]}_{|A_{kj}|} + 1$$

- But we don't know which $a_k \in S_{ij}$ to choose, so:

hk

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1 \mid a_k \in S_{ij}\} & \text{otherwise} \end{cases}$$

Observations

- You can implement this recurrence using:
  - Recursive algorithm with memoization
  - Bottom-up dynamic programming with table-filling
- But... there is a simpler and more efficient approach:

Greedy Choice

A single carefully chosen activity can reduce the problem to one smaller subproblem.

- Let's explore this next.

Why Greedy is Different from Dynamic Programming

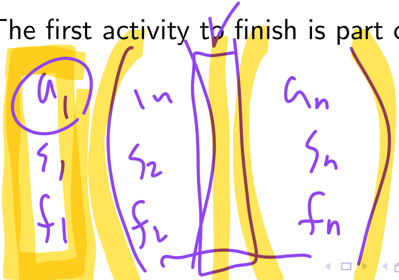
- In **dynamic programming**, you:
  - Consider all activities a_k in S_{ij}.
  - Solve all subproblems S_{ik} and S_{kj} for each a_k.
  - Then decide which a_k gives the best result.
- In the **greedy approach**, you:
  - Directly choose the activity that finishes earliest (e.g., a_1).
  - Immediately add it to your solution.
  - Then solve only one subproblem: activities starting after a_1.

Greedy Choice Strategy

- Since the activities are sorted by finish time:

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

  the greedy choice is the first activity a_1.
- If multiple activities finish at the same earliest time, choose any one.
- Key insight: The first activity to finish is part of some optimal solution.



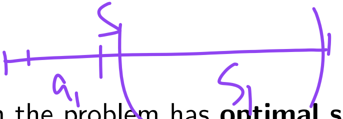
Reducing the Problem After Greedy Choice

- Once a_1 is selected, we only need to consider:

$$S_1 = \{a_i \in S \mid s_i \geq f_1\}$$

- Why not consider activities finishing before a_1?
  - Because f_1 is the earliest finish time.
  - No activity ends before s_1.
- The remaining task: solve the subproblem S_1.

Greedy Choice and Optimal Substructure



- We've already shown the problem has **optimal substructure**.
- If a_1 is part of the optimal solution, then:

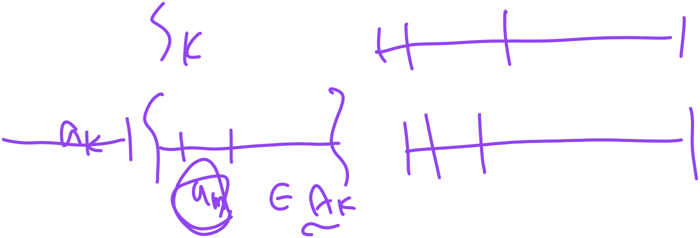
$$\text{Optimal solution} = \{a_1\} \cup \text{optimal solution for } S_1$$

- Key question: Is this greedy strategy always valid?
- Answer: Yes — shown via Theorem 15.1.

Theorem 15.1: Greedy Choice is Safe

Statement

Let S_k be any nonempty subproblem. Let a_m be the activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.



Proof Sketch of Theorem 15.1

- Let A_k be a maximum-size compatible subset of S_k.
- Let $a_j \in A_k$ be the activity with the earliest finish time.
- If $a_j = a_m$, we're done.
- Otherwise, replace a_j with a_m to form:

$a_j \neq a_m$

$$A'_k = (A_k \setminus \{a_j\}) \cup \{a_m\}$$

- A'_k is still compatible, same size as A_k, and includes a_m.

Implications of Theorem 15.1

- You don't need dynamic programming.
- Greedy algorithm:
  - Repeatedly select the earliest finishing activity.
  - Discard overlapping activities.
- Each selected activity's finish time increases strictly.
- Only one pass needed (in sorted order) $\rightarrow$ **Efficient**.

Top-Down Design of Greedy Algorithms

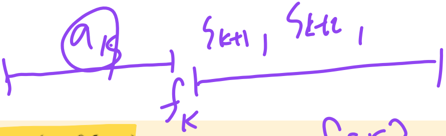
- Unlike DP, greedy algorithms use a **top-down** approach.
- Make a choice $\rightarrow$ reduce to a subproblem $\rightarrow$ repeat.
- Dynamic programming works bottom-up: solve subproblems first.
- Greedy algorithms are often simpler, faster, and easier to implement.

A Recursive Greedy Algorithm

- Instead of solving all subproblems (as in DP), we use a top-down greedy approach.
- The procedure `RECURSIVE-ACTIVITY-SELECTOR`:
  - Takes start times `s[ ]` and finish times `f[ ]` as input arrays.
  - Assumes activities are sorted by finish time: $f_1 \le f_2 \le \cdots \le f_n$.
  - Uses a fictitious activity a_0 with $f_0 = 0$.
- Initial call:

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n)
```

RECURSIVE-ACTIVITY-SELECTOR Pseudocode



$s[m] \geq f[k]$

```
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 ∅
```

How the Recursive Greedy Algorithm Works

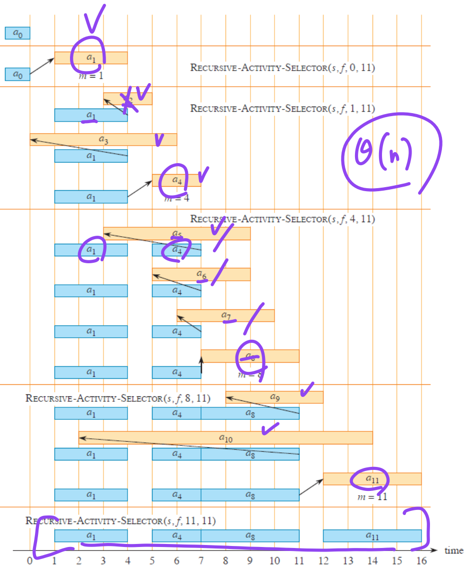
- At each call:
  - Start with activity a_k
  - Find next activity a_m such that $s[m] \geq f[k]$
- Add a_m to the result set.
- Recur on subproblem S_m (activities starting after a_m).
- Stop when no further compatible activities remain.

Efficiency:

- Each activity is examined once $\rightarrow \Theta(n)$ time (if sorted).

RECURSIVE-ACTIVITY-SELECTOR Pseudocode

i	s_i	f_i
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	7	11
9	8	12
10	2	14
11	12	16



From Recursive to Iterative Greedy Algorithm

- The recursive solution is almost **tail recursive**:
  - Recursive call is the last action in each case.
  - Many compilers can convert this to iteration automatically.
- It's straightforward to manually convert it to an **iterative** form.
- The iterative version uses a loop to simulate recursion and builds the result incrementally.

GREEDY-ACTIVITY-SELECTOR Pseudocode

GREEDY-ACTIVITY-SELECTOR(s, f, n)

```
A = {a_1}                                     {a_1}
k = 1
for m = 2 to n                  f[m]
    if s[m] ≥ f[k]                // is a_m in S_k ?
        A = A ∪ {a_m}             // yes, so choose it
        k = m                     // and continue from there
return A
```

Question?