Algorithm Analysis and Data Structures

CSCI 7432 - Fall 2022

Dynamic Programming vs Greedy Algorithms

Dr. Yao XU

Assistant Professor

Department of Computer Science

Georgia Southern University

Email: yxu@georgiasouthern.edu

Table of Contents

Dynamic Programming

Dynamic Programming

- ***Dynamic Programming (DP)*** is an algorithm design technique that typically applies to **optimization** problems.
  - Find a solution with the optimal value.
  - Minimization or maximization.
- The given problem can be defined by recurrences with overlapping subproblems.
- An **optimization problem** must have two key ingredients for **DP** to apply:
  - **Optimal substructure**
  - **Overlapping subproblems**
- **Key idea of DP:** Avoid re-computation of repeated subproblems by storing the solution to each subproblem in a table.

General Steps of Dynamic Programming

Step 1: Find a recurrence relation

- Relating the original problem's solution to solutions to its subproblems.

Step 2: Count #subproblems = #distinct recursive calls

- Subproblems overlap, but all recursive calls live in a small domain.

Step 3: Set up a DP table and store solutions to these subproblems in the table

- Solve each subproblem only once. Table size = #subproblems
- Define what each cell holds (solution value of the corresponding subproblem).
- Must make sure that when filling a cell, all values it requires have already been filled (based on recurrence relation)!

Step 4: Trace the DP table to find solution to the original problem

- Can usually be done by recursion

Dynamic Programming

The Integral Knapsack Problem

The Integral Knapsack Problem

Input:

- n items
- Item i is worth $\$v_i$, weighs w_i pounds. v_i and w_i are positive integers.
- A knapsack with capacity W, also a positive integer.

Output:

- A most valuable subset of items with total weight $\leq W$.
- ***Integral (0-1)***: Have to either take an item or not take it - can't take part of it.

Example:

- $v_1 = 6, v_2 = 10, v_3 = 12;$
- $w_1 = 1, w_2 = 2, w_3 = 3; W = 5.$

Optimal solution subset of items $= \{2, 3\}$

- total weight $= w_2 + w_3 = 5 \leq W$
- total value $= v_2 + v_3 = 22$ (maximized)

Solving Integral Knapsack (1/2)

1. Recursion

- Denote S_i as the set of the first i items, with $0 \leq i \leq n$.
- **(Optimal substructure)** Determine what to do with the **last** item
  - **Either** take it (if $W \geq w_n$), gain v_n and recurse on the remaining items in S_{n-1} with capacity $W - w_n$;
  - **Or** leave it, and recurse on the remaining item set S_{n-1} with capacity W.
- Let $V(S_n, W)$ be the total value of a most valuable subset of items selected from S_n with total weight $\leq W$.
  - **Base cases**: $V(S_n, W) = 0$, if $W = 0$ or $n = 0$.
  - If $W \geq w_n$, $V(S_n, W) = \max\{v[n] + V(S_{n-1}, W - w[n]), V(S_{n-1}, W)\}$
  - If $W < w_n$, $V(S_n, W) = V(S_{n-1}, W)$

Solving Integral Knapsack (2/2)

1. Recursion (cont'd)

- **Base cases**: $V(S_n, W) = 0$, if $W = 0$ or $n = 0$.
- If $W \geq w_n$, $V(S_n, W) = \max\{v[n] + V(S_{n-1}, W - w[n]), V(S_{n-1}, W)\}$
- If $W < w_n$, $V(S_n, W) = V(S_{n-1}, W)$

```
REC-KNAPSACK(W, w, v, n)
1  if W == 0 or n == 0
2      return 0
3  if W ≥ w[n]
4      V1 = v[n] + REC-KNAPSACK(W – w[n], w, v, n – 1)
5      V2 = REC-KNAPSACK(W, w, v, n – 1)
6      return MAX(V1, V2)
7  else
8      return REC-KNAPSACK(W, w, v, n – 1)
```

- **Q:** How many recursive calls?
- **A:** $O(2^{n+1})$
- Not too many distinct ones **(Overlapping subproblems)** —

  $(n + 1)(W + 1)$ choices overall

Integral Knapsack – A DP Solution (1/5)

2. Dynamic Programming

Step 1: Find a recurrence relation

- **Base cases**: $V(S_n, W) = 0$, if $W = 0$ or $n = 0$.
- If $W \geq w_n$, $V(S_n, W) = \max\{v[n] + V(S_{n-1}, W - w[n]), V(S_{n-1}, W)\}$
- If $W < w_n$, $V(S_n, W) = V(S_{n-1}, W)$

Step 2: Count #distinct subproblems – $(n+1)\times(W+1)$

Step 3: Define a $(n+1)\times(W+1)$ array $V[0..n][0..W]$.

- $V[i, j] = V(S_i, j)$ will hold the value (total value) of an optimal solution (a most valuable subset of items) to the subproblem defined on item set S_i (the first i items) with capacity j.
- Fill in the array according to the recurrence relation. (See next slide)
- $V[n, W]$ will be the value of an optimal solution to the original problem.

Integral Knapsack – A DP Solution (2/5)

Step 3 (cont'd): Fill in array V according to the recurrence relation:

$$V[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1,j], & \text{if } j - w_i < 0 \\ \max\{V[i-1,j], v_i + V[i-1,j-w_i]\}, & \text{if } j - w_i \geq 0 \end{cases}$$

- Cells $V[i-1,j]$ and $V[i-1,j-w_i]$ must be filled before filling $V[i,j]$.

Example:

- $v_1 = 6, v_2 = 10, v_3 = 12;$
- $w_1 = 1, w_2 = 2, w_3 = 3; W = 5.$
- Array: $V[0..3][0..5]$
- $V[3][5]$ will be the value of an optimal solution.

$V[i,j]$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					

Integral Knapsack – A DP Solution (3/5)

Step 3 (cont'd): Write pseudocode for generating array V.

```
DP-0-1-KNAPSACK(W, w, v, n)
1  let V[0..n, 0..W] be a new array
2  for j = 0 to W
3       V[0, j] = 0
4  for i = 1 to n
5       V[i, 0] = 0
6       for j = 1 to W
7            if w[i] ≤ j and v[i] + V[i - 1, j - w[i]] > V[i - 1, j]
8                 V[i, j] = v[i] + V[i - 1, j - w[i]]
9            else V[i, j] = V[i - 1, j]
10 return V
```

$$V[i,j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1,j], & \text{if } j - w_i < 0 \\ \max\{V[i-1,j], v_i + V[i-1, j-w_i]\}, & \text{if } j - w_i \geq 0 \end{cases}$$

- **Running time**: $\Theta(nW)$
- Value of an optimal solution will be $V[n, W]$.

Step 4: How to find the set of items in the optimal packing?

Integral Knapsack – A DP Solution (4/5)

Step 4: Extract an optimal solution from array V.

- Consider the last item n.
  - If $V[n, W] = V[n-1, W]$, then item n is not in the optimal solution;
  - Otherwise, item n is in the optimal solution and $V[n, W]$ is obtained from $V[n-1, W-w_n]$ by adding item n.

Example:

- $v_1 = 6, v_2 = 10, v_3 = 12;$
- $w_1 = 1, w_2 = 2, w_3 = 3; W = 5.$

$V[i,j]$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

Integral Knapsack – A DP Solution (5/5)

Step 4 (cont'd): Write pseudocode for extracting an optimal solution from V.

- Consider the last item n.
  - If $V[n, W] = V[n-1, W]$, then item n is not in the optimal solution;
  - Otherwise, item n is in the optimal solution and $V[n, W]$ is obtained from $V[n-1, W-w_n]$ by adding item n.

```
PRINT-OPT-KNAPSACK(V, n, W, w)
1  if n > 0 or W > 0
2      if V[n, W] = V[n - 1, W]
3          PRINT-OPT-KNAPSACK(V, n - 1, W, w)
4      else
5          PRINT-OPT-KNAPSACK(V, n - 1, W – w[n], w)
6          Print(n)
```

- **Running time**:

$$T(n) = T(n-1) + c$$

$$\Rightarrow T(n) \in O(n)$$

Overall running time: $\Theta(nW)$

- DP-0-1-KNAPSACK: $\Theta(nW)$
- PRINT-OPT-KNAPSACK: $O(n)$

A Note on Running Time

- $\Theta(nW)$ is not polynomial
- Input size for W: #binary bits $k \in \Theta(\log W)$
- Therefore, the running time in terms of n and k is:

$$T(n,k) \in \Theta(nW) = \Theta(2^k n),$$

  which is exponential in k.
- This is called ***pseudo-polynomial***.
  - Running time is polynomial in the numeric value of the input but exponential in the input size.

Greedy Approach

The Greedy Approach

- The ***Greedy Approach*** is an algorithm design technique that is applicable to **optimization** problems only.
- A solution is constructed through a sequence of ***greedy choices*** that are
  - ***Feasible*** - it has to satisfy the problem's constraints
  - ***Locally optimal*** - it has to be the best local choice among all feasible choices available on that step
  - ***Irrevocable*** - once made, it cannot be changed on subsequent steps
- A greedy algorithm doesn't always yield a (globally) optimal solution.
- A greedy algorithm works only when the problem has two properties:
  - ***Greedy-choice property***
  - ***Optimal substructure***

Correctness of A Greedy Algorithm (1/2)

To tell whether a **greedy algorithm** will solve an optimization problem, we usually prove the following two properties:

- ***Optimal substructure***
  - Whenever we make a choice, one subproblem remains and it just looks like the original problem, with same input type and same notion of optimal solution.
  - **Show that**

    OPT to original problem = Greedy choice + OPT to subproblem
- ***Greedy-choice property***
  - The greedy choice is always part of some optimal solution.
  - **Show that** there exists an optimal solution that contains the greedy choice.

Correctness of A Greedy Algorithm (2/2)

With these two properties, the greedy algorithm works:

1. We commit to a greedy choice c_1.
2. **Greedy-choice property:** ∃ an optimal solution A_1 to the problem s.t. $c_1 \in A_1$.
3. **Optimal substructure:** $A_1 - \{c_1\}$ is an optimal solution to the remaining subproblem.
4. Now we commit to a greedy choice c_2 for this subproblem.
5. **Greedy-choice property:** ∃ an optimal solution S_2 to the subproblem s.t. $c_2 \in S_2$.
   Then, $A_2 = S_2 \cup \{c_1\}$ is an optimal solution to the original problem. ($c_1, c_2 \in A_2$)
6. **Optimal substructure:** $S_2 - \{c_2\}$ is an optimal solution to the remaining subproblem.
7. Now we commit to a greedy choice c_3 for this subproblem.
8. **Greedy-choice property:** ∃ an optimal solution S_3 to the subproblem s.t. $c_3 \in S_3$.
   Then, $A_3 = S_3 \cup \{c_1, c_2\}$ is an optimal solution to the original problem. ($c_1, c_2, c_3 \in A_3$)

⋮

Greedy Approach

The Fractional Knapsack Problem

Fractional Knapsack Problem

Input:

- n items
- Item i is worth $\$v_i$, weighs w_i pounds, where v_i and w_i are integers
- A knapsack with capacity W, also an integer

Output:

- A most valuable subset of items with total weight $\leq W$.
- ***Fractional***: Can take fraction of an item.

Example:

- $v_1 = 6, v_2 = 10, v_3 = 12;$
- $w_1 = 1, w_2 = 2, w_3 = 3; W = 5.$

Optimal solution:

- Take l_i pounds of item i: $l_1 = 1, l_2 = 2, l_3 = 2$
- Total weight $= l_1 + l_2 + l_3 = 5 \leq W$
- Total value $= v_1 + v_2 + \frac{2}{3}v_3 = 24$ (maximized)

Fractional Knapsack - A Greedy Approach

- Look for a "safe" **greedy choice**.
- **Q:** What is the current best choice?
  - Take which item? Take how much of it?
- **Greedy choice:** Take the item with largest value per pound, v_i/w_i.
- Sort the items so that $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all $1 \leq i < n$.
- **Example:**

  $W = 5$.

i	1	2	3
v_i	6	10	12
w_i	1	2	3
v_i/w_i	6	5	4

Greedy solution:

- $l_1 = 1, l_2 = 2, l_3 = 2$
- Total value $= v_1 + v_2 + \frac{2}{3}v_3 = 24$

  – Optimal

- **Q:** Is this a "safe" greedy choice?

Is This Greedy Approach Correct?

- **Greedy approach**: Start picking items by largest value per pound, v_i/w_i, continue as long as the knapsack isn't full.
- Let $L = \langle l_1, l_2, \cdots, l_n \rangle$ be the solution found by this greedy approach. (We take l_i pounds of item i.)
- If $OPT = \langle l_1^*, l_2^*, \cdots, l_n^* \rangle$ is an optimal solution (take l_i^* pounds of item i), then we need to show that

$$\sum_{i=1}^{n} l_i \cdot \frac{v_i}{w_i} = \sum_{i=1}^{n} l_i^* \cdot \frac{v_i}{w_i}$$

Correctness of The Greedy Approach (1/3)

1. Optimal substructure

OPT to original problem - Greedy/Any choice = OPT to subproblem

Claim: Let $OPT = \langle l_1, l_2, \cdots l_n \rangle$ be an optimal solution to the original problem, where l_i is the amount of item i to be picked. Let j be any item. Then, $OPT\_{j} = \langle l_1, \cdots, l_{j-1}, l_{j+1}, \cdots, l_n \rangle$ must be an optimal solution to the subproblem defined on all items excluding j with knapsack capacity $W - l_j$.

Proof. (by contradiction)

- Assume OPT_{j} is not optimal to the subproblem.
- Then, there must be an OPT'_{j} with total value greater than that of OPT_{j}.
- Thus, for the original problem, OPT'_{j} together with l_j is also feasible and the total value is greater than that of OPT.

$\Rightarrow OPT$ is not optimal – A contradiction. □

Correctness of The Greedy Approach (2/3)

2. Greedy-choice property

Assume the items are sorted s.t. $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$.

Claim: There exists an optimal solution where item 1 is ***saturated*** – we cannot pick anymore of item 1, i.e., $l_1 = \min\{W, w_1\}$.

Proof. Let $OPT = \langle l_1, l_2, \cdots l_n \rangle$ be an optimal solution to the original problem. We will show that if $l_1 < \min\{W, w_1\}$, we can alter OPT to obtain another optimal solution $OPT' = \langle l'_1, l'_2, \cdots, l'_n \rangle$ with $l'_1 = \min\{W, w_1\}$.

- **Observation:** We must have $\sum_i l_i = W$. - Why?
- Let $\Delta = \min\{W, w_1\} - l_1$.
- We alter the solution OPT by
  - taking Δ more of item 1: $l'_1 = l_1 + \Delta$
  - taking Δ less of other item(s): $\sum_{2 \leq i \leq n} l'_i = \sum_{2 \leq i \leq n} l_i - \Delta$

    - Need to show $\sum_{2 \leq i \leq n} l_i \geq \Delta$ for this change to be valid.

Correctness of The Greedy Approach (3/3)

2. Greedy-choice property (cont'd)

Assume the items are sorted s.t. $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$.

Claim: There exists an optimal solution where item 1 is ***saturated*** – we cannot pick anymore of item 1, i.e., $l_1 = \min\{W, w_1\}$.

Proof. (cont'd)

- We alter the solution OPT by
  - taking Δ more of item 1: $l_1' = l_1 + \Delta$
  - taking Δ less of other item(s): $\sum_{2 \leq i \leq n} l_i' = \sum_{2 \leq i \leq n} l_i - \Delta$
- This will not decrease the total value:

$$\sum_i \left(l_i' \cdot \frac{v_i}{w_i} \right) \geq \sum_i \left(l_i \cdot \frac{v_i}{w_i} \right)$$

as v_1/w_1 is the largest. □

Fractional Knapsack - A Greedy Algorithm

- **Greedy approach**: Start picking items by largest value per pound, v_i/w_i, continue as long as the knapsack isn't full.
- First, sort the items so that $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all $1 \leq i < n$.

```
FRACTIONAL-KNAPSACK(W, w, v, n)
1  let L[1..n] be a new array
2  load = 0
3  i = 1
4  while load < W and i ≤ n
5      if w[i] ≤ W – load
6          L[i] = w[i]
7      else L[i] = W – load
8      load = load + w[i]
9      i = i + 1
10 return L
```

- **Example:**

$W = 5$

i	1	2	3
v_i	6	10	12
w_i	1	2	3
v_i/w_i	6	5	4

- **Solution:** $L = \langle 1, 2, 2 \rangle$
- **Overall running time**: $O(n \log n)$
  - Sorting takes $O(n \log n)$ time
  - FRACTIONAL-KNAPSACK takes $O(n)$ time

Greedy Approach

Not Work for Integral Knapsack

Greedy Approach Not Work for Integral Knapsack

- Consider the **greedy approach**: start picking the items by largest value per pound, v_i/w_i, continue as long as the knapsack isn't full.
- **Example:**

$W = 5$.

i	1	2	3
v_i	6	10	12
w_i	1	2	3
v_i/w_i	6	5	4

- **Greedy solution:** Take items 1 and 2

  Total value $= v_1 + v_2 = 16$
- **Optimal solution:** Take items 2 and 3

  Total value $= v_2 + v_3 = 22$

To show that a greedy algorithm does not always yield an optimal solution, we usually find a *counterexample* (an instance) for which

- The algorithm fails to produce an optimal solution (like above).
- Or, we find an optimal solution and show that it can never be altered to include our greedy choice without changing the optimal value.

Dynamic Programming vs Greedy Approach

Dynamic Programming vs Greedy Approach

Dynamic Programming

- Two key ingredients for applying DP to solve an optimization problem: **optimal substructure** and **overlapping subproblems**
- Solves all dependent subproblems first and then makes a choice that will lead to an optimal solution;
- Always yields an optimal solution;
- Is usually less efficient as compared to a greedy algorithm.

Greedy Approach

- Makes a locally optimal choice at every step and never look back;
- NOT always lead to an overall optimal solution;
- Yields an optimal solution iff the following two properties are satisfied (correctness proof): **greedy-choice property** and **optimal substructure**;
- Is usually an efficient algorithm.

An Activity Selection Problem

An Activity Selection Problem

Input:

- n jobs: job i has start time s_i, finish time f_i, and $0 \leq s_i \leq f_i$.
- We have one machine that can execute only one job at a time.
- Two jobs i, j are ***compatible*** if their intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

Output:

- A maximum-size subset of mutually compatible jobs.

Example 1:

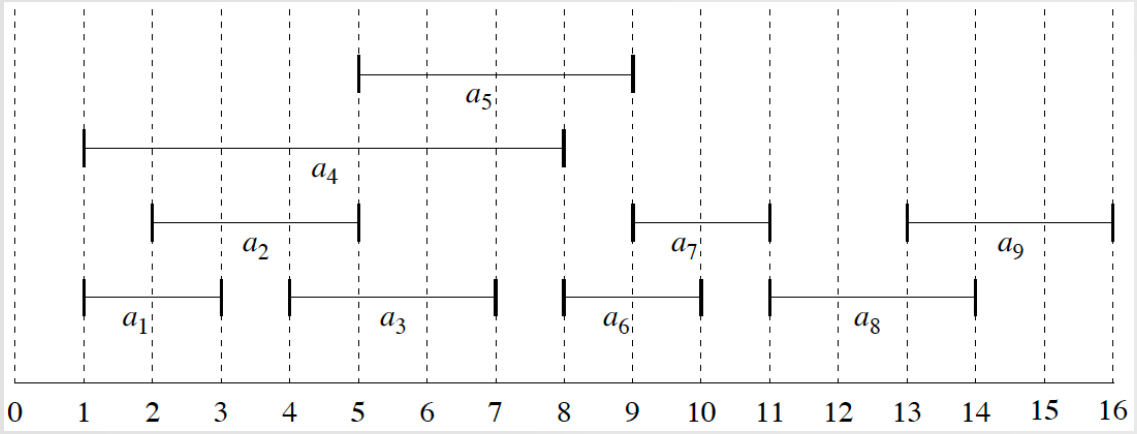
i	1	2	3	4
s_i	1	3	1	4
f_i	3	5	4	5

Optimal solutions: $\{1, 2\}, \{1, 4\}, \{3, 4\}$

Another Example

Example 2:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Optimal solutions:

- $\{a_1, a_3, a_6, a_8\}$
- $\{a_1, a_3, a_6, a_9\}$
- $\{a_1, a_3, a_7, a_8\}$
- $\{a_1, a_3, a_7, a_9\}$
- $\{a_1, a_5, a_7, a_8\}$
- $\{a_1, a_5, a_7, a_9\}$
- $\{a_2, a_5, a_7, a_8\}$
- $\{a_2, a_5, a_7, a_9\}$

Q: Is there a **greedy algorithm** that always produces an optimal solution?

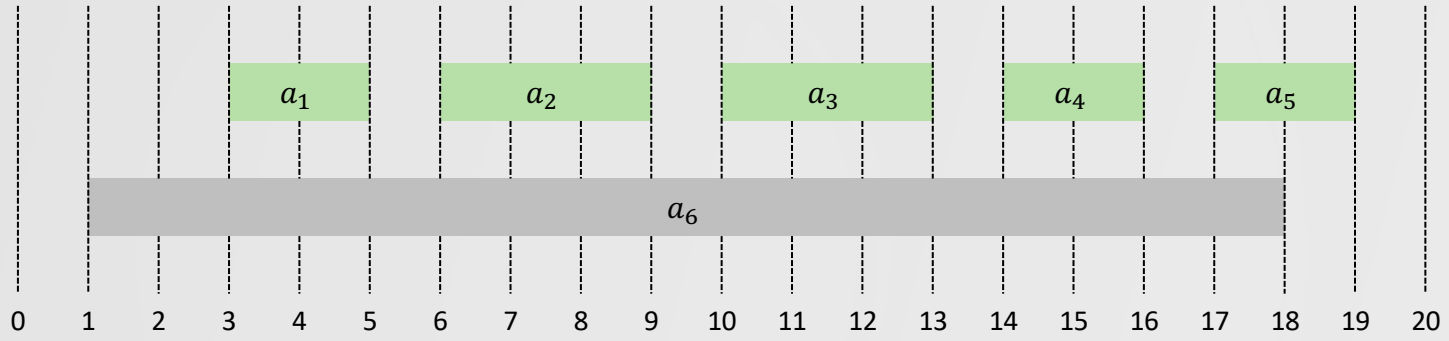
An Activity Selection Problem

A Greedy Algorithm

A Greedy Choice (1/4)

- Is there always a "safe" greedy choice?
- Possible greedy choices:
  1) Earliest start time? ⟵ NOT a "safe" greedy choice!

Counterexample:

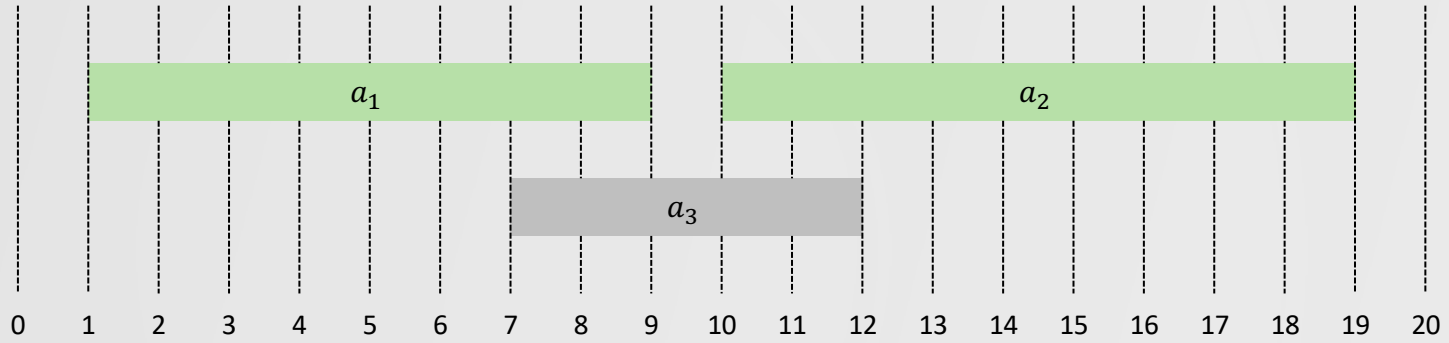


- **Greedy solution:** $\{a_6\}$ – NOT optimal
- **Optimal solution:** $\{a_1, a_2, a_3, a_4, a_5\}$

A Greedy Choice (2/4)

- Is there always a "safe" greedy choice?
- Possible greedy choices:
  2) Smallest processing time $(f - s)$? ⟵ NOT a "safe" greedy choice!

Counterexample:

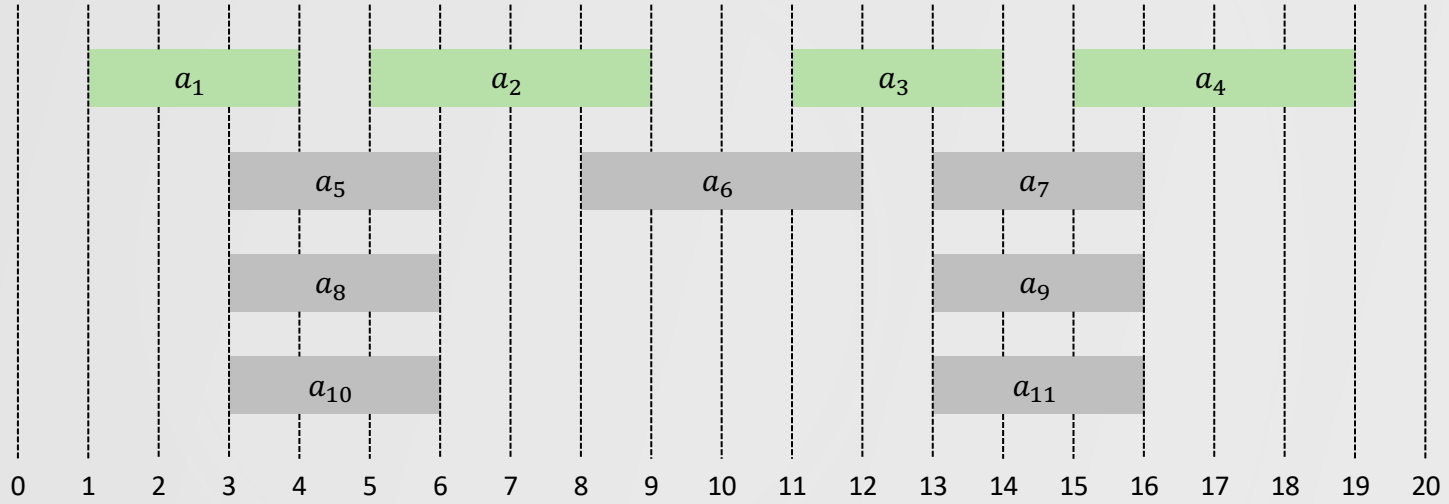


- **Greedy solution:** $\{a_3\}$ – NOT optimal
- **Optimal solution:** $\{a_1, a_2\}$

A Greedy Choice (3/4)

- Is there always a "safe" greedy choice?
- Possible greedy choices:
  3) Overlaps the fewest other remaining jobs? ⟵ NOT a "safe" greedy choice!

Counterexample:

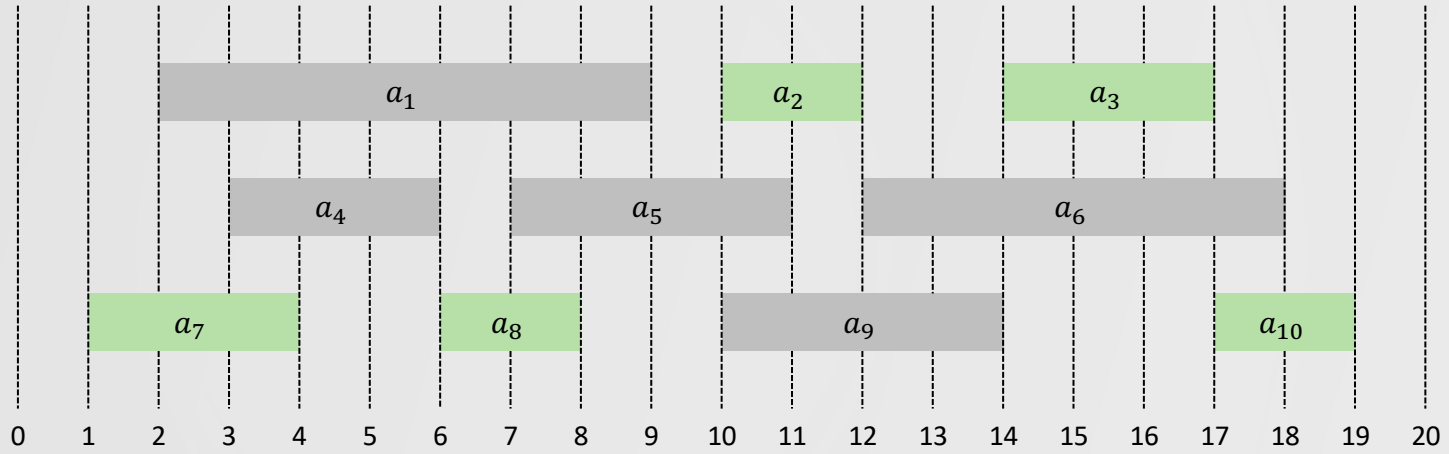


- **Greedy solution:** $\{a_6, a_1, a_4\}$ – NOT optimal
- **Optimal solution:** $\{a_1, a_2, a_3, a_4\}$

A Greedy Choice (4/4)

- Is there always a "safe" greedy choice?
- Possible greedy choices:
  4) Earliest finish time? ⟵ This is a "safe" greedy choice!

Example:



- **Greedy solution:** $\{a_7, a_8, a_2, a_3, a_{10}\}$ – Optimal

No counterexamples can be found – Need to prove correctness!

Is This Greedy Algorithm Correct?

- A **greedy algorithm**:
  1. Select a job with earliest finish time.
  2. Remove all jobs that overlap with the selected job.
  3. Repeat steps 1 and 2 until there is no job left.
- Let A be the set of jobs selected by this greedy algorithm.
- If OPT is an optimal solution (maximum-size set of mutually compatible jobs), then $A = OPT$?
  - Not likely!
  - Instead, we will show that $|A| = |OPT|$.

Proving Optimality (1/3)

First, show that our greedy choice is always part of some optimal solution.

Theorem. There exists an optimal solution that contains the job with the earliest finish time. *

- Re-arrange the jobs so that $f_1 \leq f_2 \leq \cdots \leq f_n$.

Proof. Consider an arbitrary optimal solution OPT.

- If $a_1 \in OPT$, we are done.
- If $a_1 \notin OPT$, there is a job $a_j \in OPT$ such that replacing a_j by a_1 also results in an optimal solution. (Prove in the next slide)

** See the complete proof of Theorem 16.1 on p.481 of the textbook.*

Proving Optimality (2/3)

Theorem. There exists an optimal solution that contains the job with the earliest finish time.

Proof. (cont'd)

If $a_1 \notin OPT$, let $a_j \in OPT$ be the job with earliest finish time in OPT.

Claim: a_j must be the only job in OPT that overlaps with a_1. (Prove in the next slide)

- Let $OPT' = \left(OPT - \{a_j\}\right) \cup \{a_1\}$.
- The claim implies that
  - OPT' is feasible (no overlapping jobs)
  - $|OPT'| = |OPT|$.

□

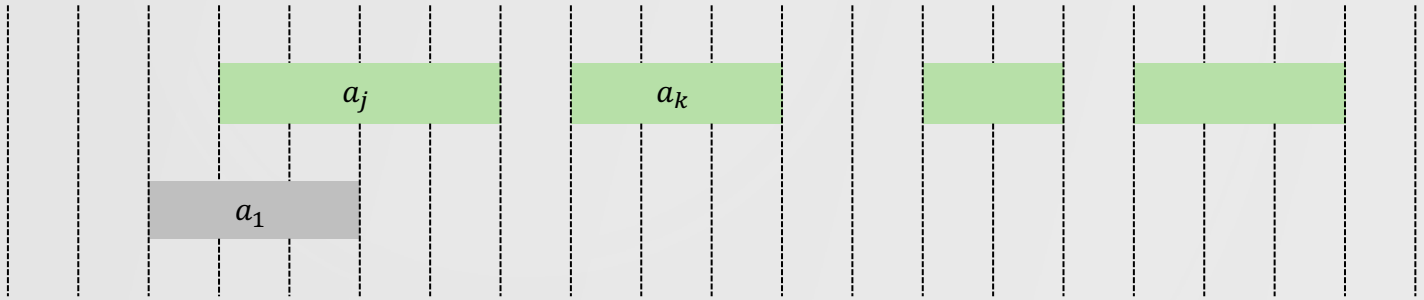
Proving Optimality (3/3)

If $a_1 \notin OPT$, let $a_j \in OPT$ be the job with earliest finish time in OPT.

Claim: a_j must be the only job in OPT that overlaps with a_1.

Proof.

- Every job $a_k \in OPT - \{a_j\}$ is compatible with a_1.
  - $f_k > f_1 \Rightarrow f_1 \leq f_j < s_k < f_k$
- a_1 and a_j must overlap.
  - Otherwise, OPT is not optimal.



□

Activity Selection – Greedy Algorithm Correctness

- **Greedy-choice property**

  Theorem. There exists an optimal solution that contains the job with the earliest finish time. (Proved on slides 12-14.)

- **Optimal substructure**

  Claim: Let OPT be an optimal solution to the original problem and let a_j be any job in OPT. Then $OPT - \{a_j\}$ is an optimal solution to the subproblem defined on all jobs that are compatible with a_j.

  Proof. (by contradiction) Assume $OPT - \{a_j\}$ is not optimal to the subproblem.

  - S is an optimal solution to the subproblem.
  - $|S| > |OPT| - 1$ and all jobs in S are compatible with a_j.
  - Thus, $OPT' = S \cup \{a_j\}$ is feasible and $|OPT'| > |OPT|$.

    $\Rightarrow OPT$ is not optimal – A contradiction. □

A Recursive Greedy Algorithm

- First, re-arrange the jobs so that $f_1 \leq f_2 \leq \cdots \leq f_n$.
- The following recursive greedy algorithm solves a subproblem defined on the job set $A_k = \{a_{k+1}, a_{k+2}, \cdots, a_n\}$.

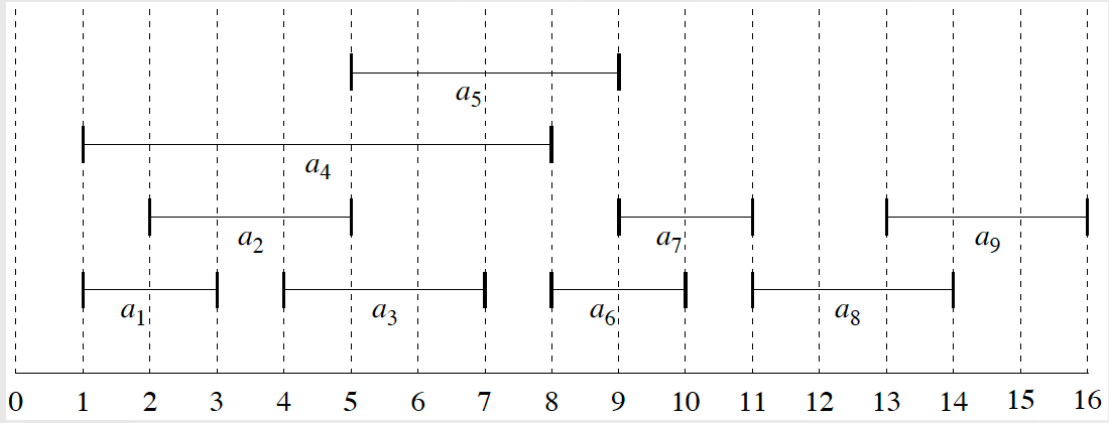
```
REC-ACTIVITY-SELECTOR(s, f, k, n)
1  i = k + 1
2  while i ≤ n and s[i] < f[k]
3       i = i + 1
4  if i ≤ n
5       return {a_i} ∪ REC-ACTIVITY-SELECTOR(s, f, i, n)
6  else
7       return ∅
```

Initial call:

REC-ACTIVITY-SELECTOR$(s, f, 0, n)$

Example:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Returned optimal solution: $\{a_1, a_3, a_6, a_8\}$

Running Time of Recursive Greedy

```
REC-ACTIVITY-SELECTOR(s, f, k, n)
1  i = k + 1
2  while i ≤ n and s[i] < f[k]
3       i = i + 1
4  if i ≤ n
5       return {a_i} ∪ REC-ACTIVITY-SELECTOR(s, f, i, n)
6  else
7       return ∅
```

Initial call: REC-ACTIVITY-SELECTOR$(s, f, 0, n)$

- Sorting takes $O(n \log n)$ time.
- REC-ACTIVITY-SELECTOR takes $O(n)$ time.
- **Total running time:** $T(n) \in O(n \log n)$

- REC-ACTIVITY-SELECTOR(s, f, k, n) solves a subproblem defined on job set $\overline{A_k} = \{a_{k+1}, a_{k+2}, \cdots, a_n\}$.
- **Running time:**
  - **while** loop: l iterations

    The first l jobs in $\overline{A_k}$ have $s < f_k$ and the next job has $s \geq f_k$.
  - $T_{Rec}(n) = O(l) + T_{Rec}(n - l)$
    $\Rightarrow T_{Rec}(n) \in O(n)$

An Iterative Greedy Algorithm

A non-recursive (iterative) version of this greedy algorithm:

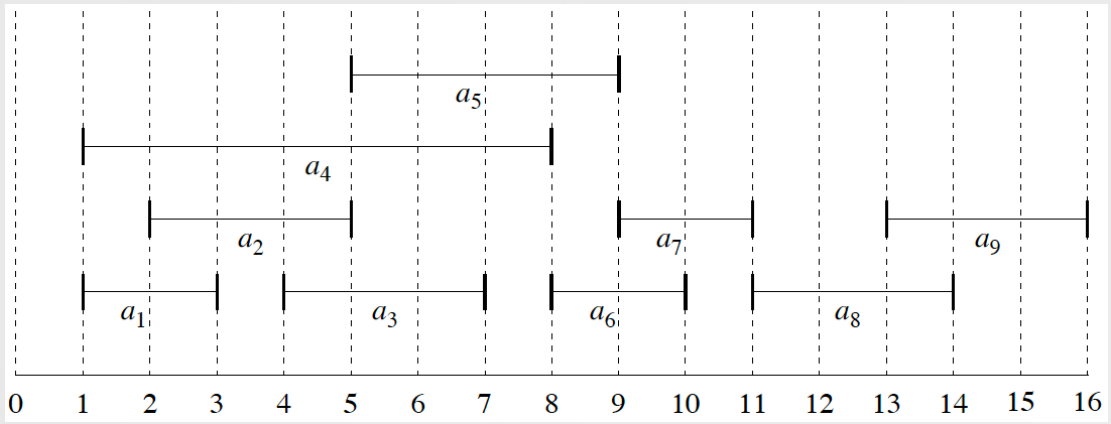
- Still, re-arrange the jobs first so that $f_1 \leq f_2 \leq \cdots \leq f_n$. – takes $O(n \log n)$ time

```
ITR-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for i = 2 to n
5      if s[i] ≥ f[k]
6          A = A ∪ {a_i}
7          k = i
8  return A
```

- Running time: $T_{Itr}(n) \in O(n)$
- **Total running time:** $T(n) \in O(n \log n)$

Example:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



Returned optimal solution: $\{a_1, a_3, a_6, a_8\}$

An Activity Selection Problem

Using Dynamic Programming

Activity Selection – A DP Solution (1/6)

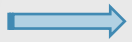
Step 1: Find a recurrence relation.

- Select job k.
- Remove all jobs incompatible with k.
- Recurse on the remaining jobs.

Jobs are re-arranged so that $f_1 \leq f_2 \leq \cdots \leq f_n$ - takes $O(n \log n)$ time.

Example:

i	1	2	3	4
s_i	1	3	1	4
f_i	3	5	4	5



i	1	2	3	4
s_i	1	1	3	4
f_i	3	4	5	5

Activity Selection – A DP Solution (2/6)

Step 1 (cont'd): Find a recurrence relation.

Denote $S_{i,j}$ as the set of jobs that start after job i finishes and that finish before job j starts.

- Create two fictious jobs: job 0 with $f_0 = 0$ and job $n + 1$ with $s_{n+1} = \infty$.
- The size of an optimal set of mutually compatible jobs in $S_{i,j}$ will be

$$OptC(S_{i,j}) = \max_{k \in S_{i,j}} \{OptC(S_{i,k}) + OptC(S_{k,j}) + 1\}.$$

- **Base cases**: $OptC(S_{i,j}) = 0$ if $S_{i,j} = \emptyset$.
- $S_{0,n+1}$ will be an optimal solution to the original problem.

Example:

i	0	1	2	3	4	5
s_i		1	1	3	4	∞
f_i	0	3	4	5	5	

Activity Selection – A DP Solution (3/6)

Step 2: Count distinct subproblems – $(n+2)\times(n+2)$

Step 3: Define an $(n+2)\times(n+2)$ array C.

- $C[i,j] = OptC(S_{i,j})$, $0 \le i, j \le n+1$, will hold the size of an optimal set of mutually compatible jobs for the subproblem defined on the job set $S_{i,j}$.
- Fill in the array C according to the following recurrence

$$C[i,j] = \begin{cases} 0, & \text{if } S_{i,j} = \emptyset \\ \max\limits_{k \in S_{i,j}} \{C[i,k] + C[k,j] + 1\}, & \text{if } S_{i,j} \neq \emptyset \end{cases}$$

- $C[0, n+1] = OptC(S_{0,n+1})$ will be the size of an optimal set of mutually compatible jobs for the original problem.

Activity Selection – A DP Solution (4/6)

$$C[i,j] = \begin{cases} 0, & \text{if } S_{i,j} = \emptyset \\ \max\limits_{k \in S_{i,j}} \{C[i,k] + C[k,j] + 1\}, & \text{if } S_{i,j} \neq \emptyset \end{cases}$$

Example:

i	0	1	2	3	4	5
s_i		1	1	3	4	∞
f_i	0	3	4	5	5	

$C[i,j]$	0	1	2	3	4	5
0	0	0				
1	-	0	0			
2	-	-	0	0		
3	-	-	-	0	0	
4	-	-	-	-	0	0
5	-	-	-	-	-	0

Activity Selection – A DP Solution (5/6)

```
DP-ACTIVITY-SELECTION(s, f, n)
let C[0..n+1, 0..n+1] and J[0..n+1, 0..n+1] be new arrays
for i = 0 to n
    C[i, i] = 0
    C[i, i + 1] = 0
C[n + 1, n + 1] = 0
for c = 2 to n + 1
    for i = 0 to n – c + 1
        j = i + c
        C[i, j] = 0
        k = j - 1
        while f [i] < f [k]
            if f [i] ≤ s[k] and f [k] ≤ s[j] and C[i, k] + C[k, j] + 1 > C[i, j]
                C[i, j] = C[i, k] + C[k, j] + 1
                J[i, j] = k
            k = k – 1
PRINT-ACTIVITIES(C, J, 0, n + 1)
```

$C[i,j]$	0	1	2	3	4	5
0	0	0				
1	-	0	0			
2	-	-	0	0		
3	-	-	-	0	0	
4	-	-	-	-	0	0
5	-	-	-	-	-	0

Step 3 (cont'd): Pseudocode for generating the DP table.

- **Running time**: $O(n^3)$

Activity Selection – A DP Solution (6/6)

Step 4: Trace arrays C and J to find an optimal set of mutually compatible jobs.

- **Example:**

i	1	2	3	4
s_i	1	1	3	4
f_i	3	4	5	5

$C[i,j]$	0	1	2	3	4	5
0	0	0	0	1	1	2
1	-	0	0	0	0	1
2	-	-	0	0	0	1
3	-	-	-	0	0	0
4	-	-	-	-	0	0
5	-	-	-	-	-	0

```
PRINT-ACTIVITIES(C, J, i, j)
if C[i, j] > 0
    k = J[i, j]
    PRINT-ACTIVITIES(C, J, i, k)
    Print(k)
    PRINT-ACTIVITIES(C, J, k, j)
```

- **Running time**: $O(n)$
- **Overall running time** (including sorting): $O(n^3)$

Thank you!
Questions?