Chapter 16

Greedy Algorithms

Greedy Algorithms

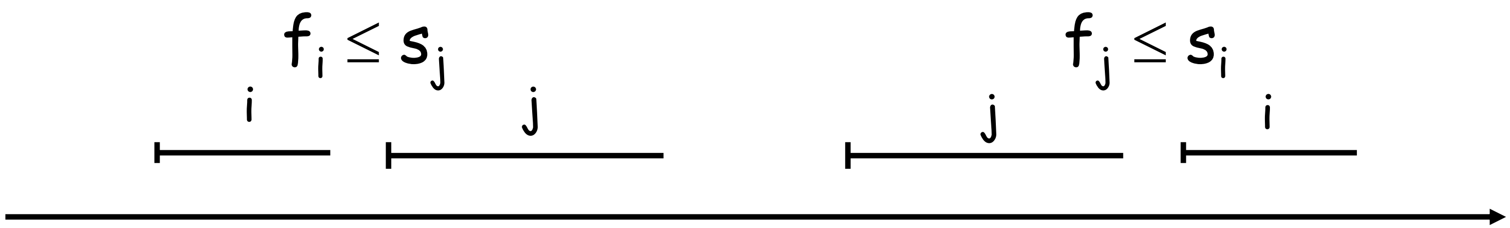
- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems
- **Idea:** When we have a choice to make, make the one that looks best right now
  - Make a locally optimal choice in hope of getting a globally optimal solution
- Greedy algorithms don't always yield an optimal solution
- When the problem has certain general characteristics, greedy algorithms give optimal solutions

Activity Selection

- Schedule n **activities** that require exclusive use of a common resource

  $S = \{a_1, \ldots, a_n\}$ – set of activities

- a_i needs resource during period $[s_i, f_i)$
  - s_i = **start time** and f_i = **finish time** of activity a_i
  - $0 \leq s_i < f_i < \infty$
- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap



Activity Selection Problem

Select the largest possible set of nonoverlapping (***mutually compatible***) activities.

E.g.:

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

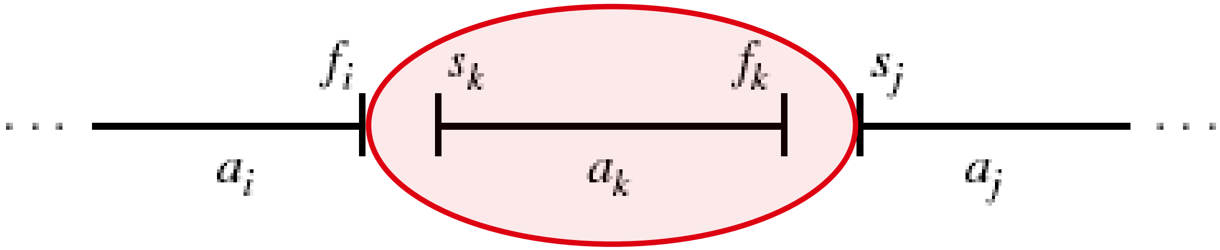
- Activities are sorted in increasing order of finish times
- A subset of mutually compatible activities: $\{a_3, a_9, a_{11}\}$
- Maximal set of mutually compatible activities: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$

Optimal Substructure

- Define the space of subproblems:

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

  – activities that start after a_i finishes and finish before a_j starts



- Activities that are compatible with the ones in S_{ij}
  – All activities that finish by f_i
  – All activities that start no earlier than s_j

Representing the Problem

- Add fictitious activities
  - $a_0 = [-\infty, 0)$
  - $a_{n+1} = [\infty, \text{"}\infty + 1\text{"})$
  - Range for S_{ij} is $0 \leq i, j \leq n + 1$

$S = S_{0,n+1}$ entire space of activities

- In a set S_{ij} we assume that activities are sorted in increasing order of finish times:

$$f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n < f_{n+1}$$

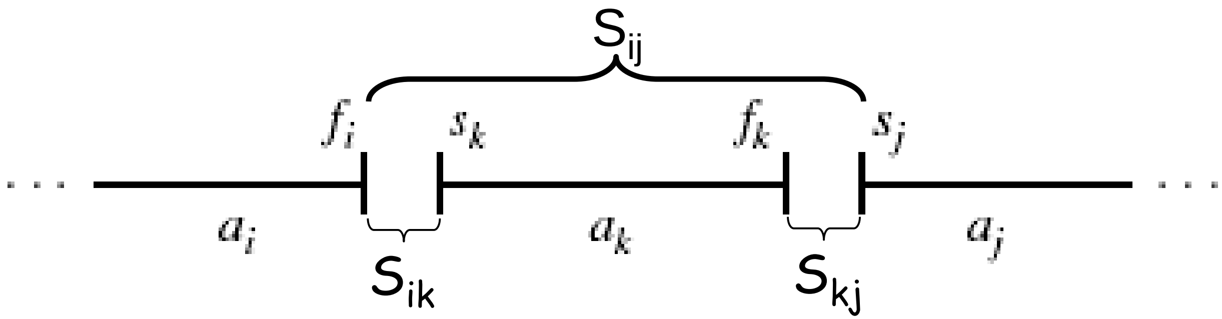
- What happens if $i \geq j$?
  - For an activity $a_k \in S_{ij}$: $f_i \leq s_k < f_k \leq s_j < f_j$

    contradiction with $f_i \geq f_j$!

  $\Rightarrow S_{ij} = \varnothing$ (the set S_{ij} must be empty!)
- We only need to consider sets S_{ij} with $0 \leq i < j \leq n + 1$

Optimal Substructure

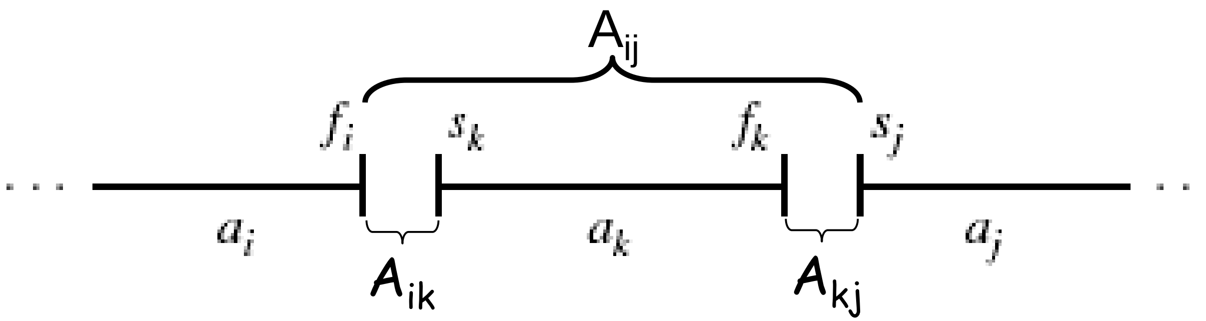
- Subproblem:
  - Select a maximum size subset of mutually compatible activities from set S_{ij}
- Assume that a solution to the above subproblem includes activity a_k (S_{ij} is non-empty)



Solution to S_{ij} = (Solution to S_{ik}) $\cup$ $\{a_k\}$ $\cup$ (Solution to S_{kj})

$|$ Solution to S_{ij} $|$ = $|$ Solution to S_{ik} $|$ + 1 + $|$ Solution to S_{kj} $|$

Optimal Substructure (cont.)



A_{ij} = Optimal solution to S_{ij}

- **Claim**: Sets A_{ik} and A_{kj} must be optimal solutions
- Assume $\exists$ A_{ik}' that includes more activities than A_{ik}

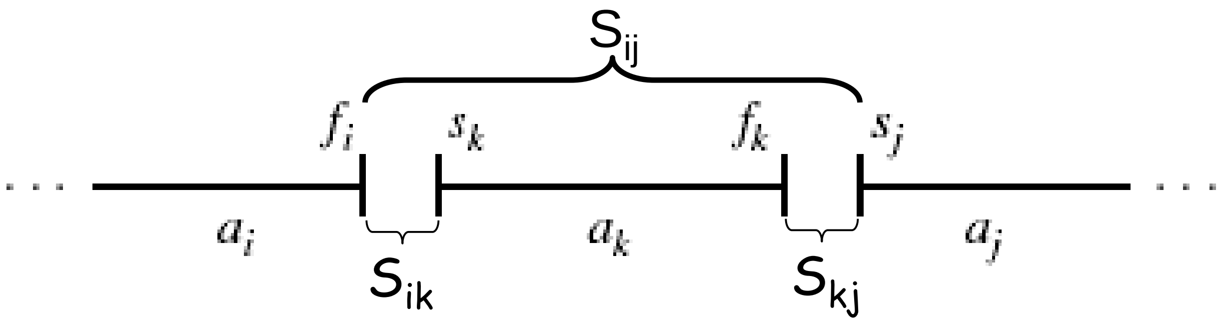
$$Size[A_{ij}'] = Size[A_{ik}'] + 1 + Size[A_{kj}] > Size[A_{ij}]$$

$\Rightarrow$ Contradiction: we assumed that A_{ij} is the maximum # of activities taken from S_{ij}

Recursive Solution

- Any optimal solution (associated with a set S_{ij}) contains within it optimal solutions to subproblems S_{ik} and S_{kj}
- c[i, j] = size of maximum-size subset of mutually compatible activities in S_{ij}
- If $S_{ij} = \varnothing \Rightarrow c[i, j] = 0$ $(i \geq j)$

Recursive Solution



If $S_{ij} \neq \varnothing$ and if we consider that a_k is used in an optimal solution (maximum-size subset of mutually compatible activities of S_{ij})

$$c[i, j] = c[i,k] + c[k, j] + 1$$

Recursive Solution

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max\limits_{\substack{i < k < j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \varnothing \end{cases}$$

- There are $j - i - 1$ possible values for k
  - $k = i+1, \ldots, j - 1$
  - a_k cannot be a_i or a_j (from the definition of S_{ij})

    $S_{ij} = \{ a_k \in S : f_i \le s_k < f_k \le s_j \}$
  - We check all the values and take the best one

We could now write a dynamic programming algorithm

Theorem

Let $S_{ij} \neq \emptyset$ and a_m the activity in S_{ij} with the earliest finish time:

$$f_m = \min \{ f_k : a_k \in S_{ij} \}$$

Then:

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}
   – There exists some optimal solution that contains a_m
2. $S_{im} = \emptyset$
   – Choosing a_m leaves S_{mj} the only nonempty subproblem

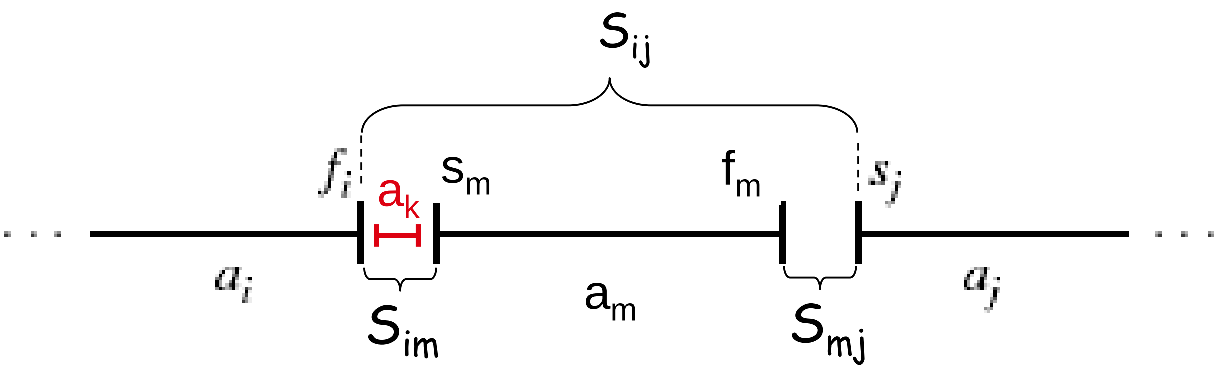
Proof

2. Assume $\exists\, a_k \in S_{im}$

   $f_i \leq s_k < f_k \leq s_m < f_m$

$\Rightarrow$ $f_k < f_m$ contradiction !

   a_m did not have the earliest finish time

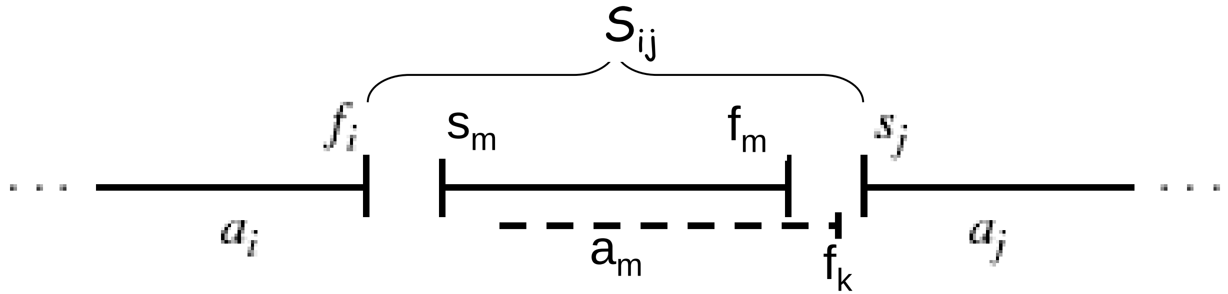


$\Rightarrow$ There is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$

Proof : Greedy Choice Property

1. a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}

- A_{ij} = optimal solution for activity selection from S_{ij}
  - Order activities in A_{ij} in increasing order of finish time
  - Let a_k be the first activity in $A_{ij} = \{a_k, \ldots\}$
- If $a_k = a_m$ Done!
- Otherwise, replace a_k with a_m (resulting in a set A_{ij}')
  - since $f_m \leq f_k$ the activities in A_{ij}' will continue to be compatible
  - A_{ij}' will have the same size with $A_{ij} \Rightarrow a_m$ is used in some maximum-size subset



Why is the Theorem Useful?

	Dynamic programming	Using the theorem
Number of subproblems in the optimal solution	2 subproblems: S_{ik}, S_{kj}	1 subproblem: S_{mj} $S_{im} = \varnothing$
Number of choices to consider	$j - i - 1$ choices	1 choice: the activity with the earliest finish time in S_{ij}

- Making the greedy choice (the activity with the earliest finish time in S_{ij})
  - Reduce the number of subproblems and choices
  - Solve each subproblem in a top-down fashion

Greedy Approach

- To select a maximum size subset of mutually compatible activities from set S_{ij}:
  - Choose $a_m \in S_{ij}$ with earliest finish time (greedy choice)
  - Add a_m to the set of activities used in the optimal solution
  - Solve the same problem for the set S_{mj}
- From the theorem
  - By choosing a_m we are guaranteed to have used an activity included in an optimal solution

    $\Rightarrow$ We do not need to solve the subproblem S_{mj} before making the choice!
  - The problem has the **GREEDY CHOICE** property

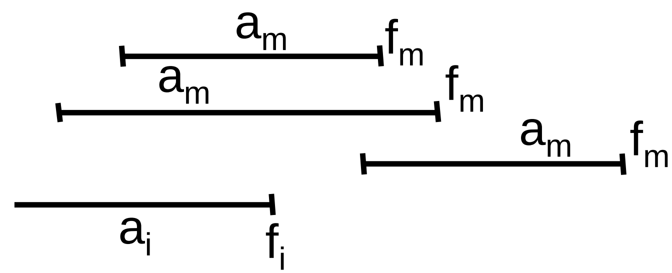
Characterizing the Subproblems

- The original problem: find the maximum subset of mutually compatible activities for $S = S_{0, n+1}$
- Activities are sorted by increasing finish time

$$a_0, a_1, a_2, a_3, \ldots, a_{n+1}$$

- We always choose an activity with the earliest finish time
  - Greedy choice maximizes the unscheduled time remaining
  - Finish time of activities selected is strictly increasing

A Recursive Greedy Algorithm



Alg.: REC-ACT-SEL (s, f, i, n)

$m \leftarrow i + 1$

while $m \le n$ **and** $s_m < f_i$ ▸Find first activity in $S_{i,n+1}$

        do $m \leftarrow m + 1$

if $m \le n$

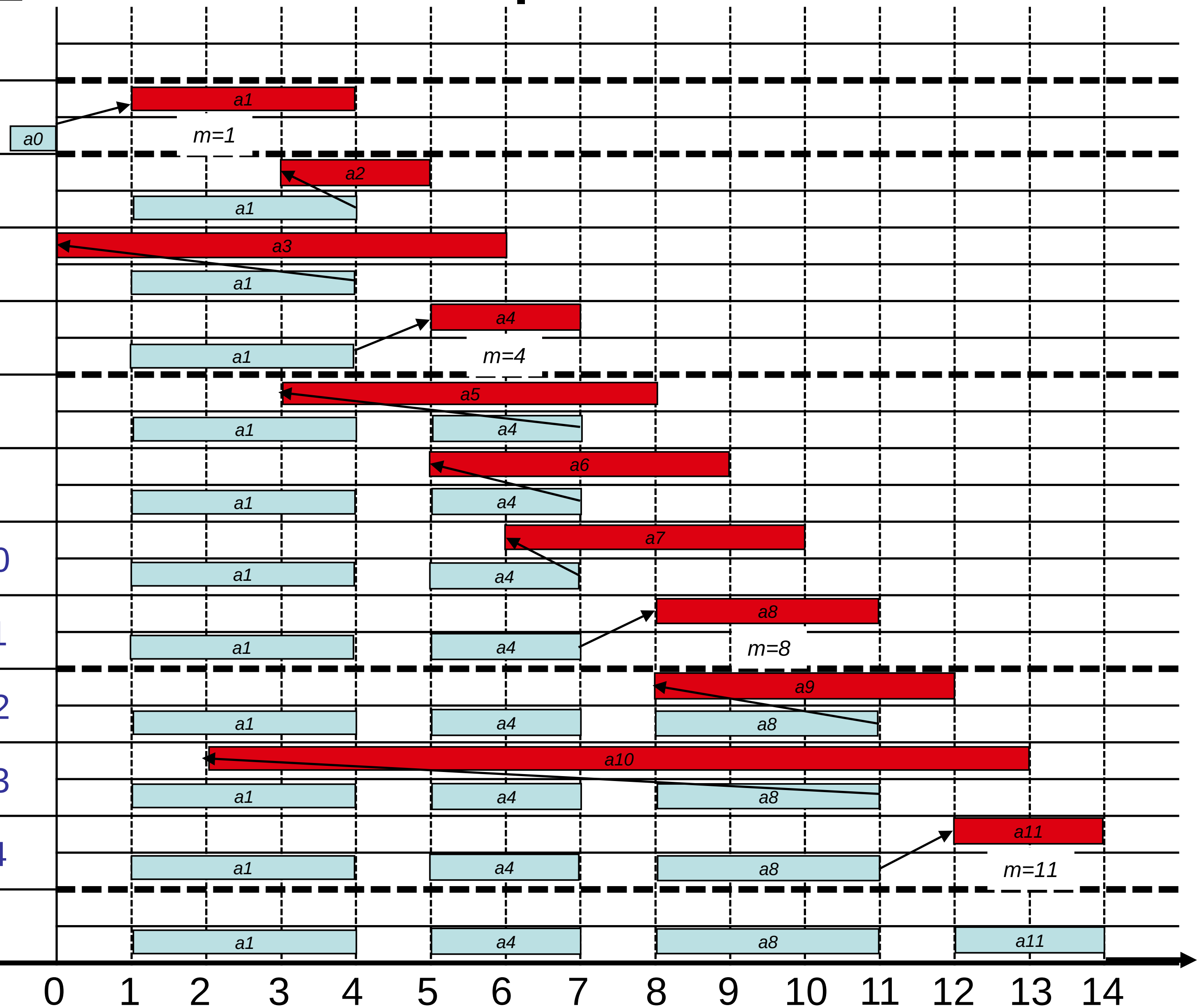
    then return $\{a_m\} \cup$ REC-ACT-SEL(s, f, m, n)

else return $\varnothing$

- Activities are ordered in increasing order of finish time
- Running time: $\Theta(n)$ – each activity is examined only once
- **Initial call:** REC-ACT-SEL(s, f, 0, n)

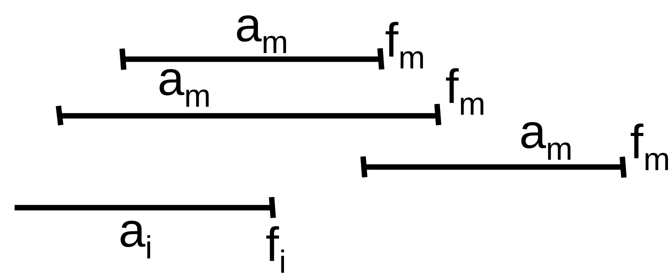
Example

k	s_k	f_k
0	-	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14
12	∞	-



An Incremental Algorithm

Alg.: GREEDY-ACTIVITY-SELECTOR(s, f, n)



$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for $m \leftarrow 2$ **to** n

  do if $s_m \geq f_i$ ▸ activity a_m is compatible with a_i

    then $A \leftarrow A \cup \{a_m\}$

      $i \leftarrow m$ ▸ a_i is most recent addition to A

return A

- Assumes that activities are ordered in increasing order of finish time
- Running time: $\Theta(n)$ – each activity is examined only once

Steps Toward Our Greedy Solution

1. Determined the optimal substructure of the problem
2. Developed a recursive solution
3. Proved that one of the optimal choices is the greedy choice
4. Showed that all but one of the subproblems resulted by making the greedy choice are empty
5. Developed a recursive algorithm that implements the greedy strategy
6. Converted the recursive algorithm to an iterative one

Designing Greedy Algorithms

1. Cast the optimization problem as one for which:

   we make a choice and are left with only one subproblem to solve

2. Prove that there is always an optimal solution to the original problem that makes the greedy choice
   – Making the greedy choice is always safe

3. Demonstrate that after making the greedy choice:

   the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

Correctness of Greedy Algorithms

1. Greedy Choice Property
   – A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
2. Optimal Substructure Property
   – We know that we have arrived at a subproblem by making a greedy choice
   – Optimal solution to subproblem + greedy choice $\Rightarrow$ optimal solution for the original problem

Activity Selection

- Greedy Choice Property

There exists an optimal solution that includes the greedy choice:

  - The activity a_m with the earliest finish time in S_{ij}

- Optimal Substructure:

If an optimal solution to subproblem S_{ij} includes activity $a_k \Rightarrow$ it must contain optimal solutions to S_{ik} and S_{kj}

Similarly, a_m + optimal solution to $S_{im} \Rightarrow$ optimal sol.

Dynamic Programming vs. Greedy Algorithms

- Dynamic programming
  - We make a choice at each step
  - The choice depends on solutions to subproblems
  - Bottom up solution, from smaller to larger subproblems
- Greedy algorithm
  - Make the greedy choice and THEN
  - Solve the subproblem arising after the choice is made
  - The choice we make may depend on previous choices, but not on solutions to subproblems
  - Top down solution, problems decrease in size

The Knapsack Problem

- **The 0-1 knapsack problem**
  - A thief rubbing a store finds n items: the i-th item is worth v_i dollars and weights w_i pounds (v_i, w_i integers)
  - The thief can only carry W pounds in his knapsack
  - Items must be taken entirely or left behind
  - Which items should the thief take to maximize the value of his load?
- **The fractional knapsack problem**
  - Similar to above
  - The thief can take fractions of items

Fractional Knapsack Problem

- Knapsack capacity: W
- There are n items: the i-th item has value v_i and weight w_i
- Goal:
  - find x_i such that for all $0 \leq x_i \leq 1$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

Fractional Knapsack Problem

- Greedy strategy 1:
  - Pick the item with the maximum value
- *E.g.:*
  - $W = 1$
  - $w_1 = 100$, $v_1 = 2$
  - $w_2 = 1$, $v_2 = 1$
  - Taking from the item with the maximum value:

    Total value taken = $v_1/w_1 = 2/100$
  - Smaller than what the thief can take if choosing the other item

    Total value (choose item 2) = $v_2/w_2 = 1$

Fractional Knapsack Problem

Greedy strategy 2:

- Pick the item with the maximum value per pound v_i/w_i
- If the supply of that element is exhausted and the thief can carry more: take as much as possible from the item with the next greatest value per pound
- It is good to order items based on their value per pound

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Fractional Knapsack Problem

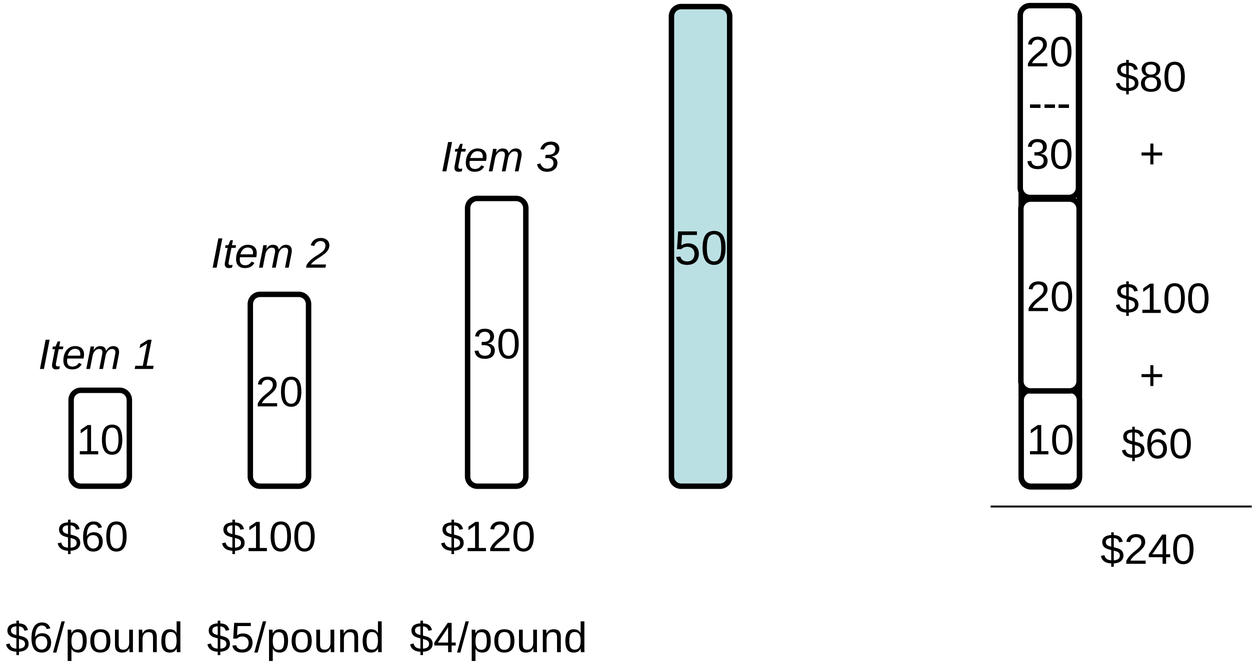
Alg.: Fractional-Knapsack (W, $v[n]$, $w[n]$)

1. While $w > 0$ and as long as there are items remaining
2. pick item with maximum v_i/w_i
3. $x_i \leftarrow \min(1, w/w_i)$
4. remove item i from list
5. $w \leftarrow w - x_i w_i$

- w – the amount of space remaining in the knapsack ($w = W$)
- Running time: $\Theta(n)$ if items already ordered; else $\Theta(n\lg n)$

Fractional Knapsack - Example

- *E.g.:*



Greedy Choice

Items:	1	2	3	…	j	…	n
Optimal solution:	x_1	x_2	x_3		x_j		x_n
Greedy solution:	x_1'	x_2'	x_3'		x_j'		x_n'

- We know that: $x_1' \geq x_1$
  - greedy choice takes as much as possible from item 1
- Modify the optimal solution to take x_1' of item 1
  - We have to decrease the quantity taken from some item j: the new x_j is decreased by: $(x_1' - x_1)\, w_1/w_j$
- Increase in profit: $(x_1' - x_1)\, v_1$
- Decrease in profit: $(x_1' - x_1) w_1\, v_j/w_j$

$$(x_1' - x_1)\, v_1 \geq (x_1' - x_1) w_1\, v_j/w_j$$

$$v_1 \geq w_1 \frac{v_j}{w_j} \quad \Rightarrow \quad \frac{v_1}{w_1} \geq \frac{v_j}{w_j}$$

True, since x_1 had the best value/pound ratio

Optimal Substructure

- Consider the most valuable load that weights at most W pounds
- If we remove a weight w of item j from the optimal load

$\Rightarrow$ The remaining load must be the most valuable load weighing at most $W - w$ that can be taken from the remaining $n - 1$ items plus $w_j - w$ pounds of item j

The 0-1 Knapsack Problem

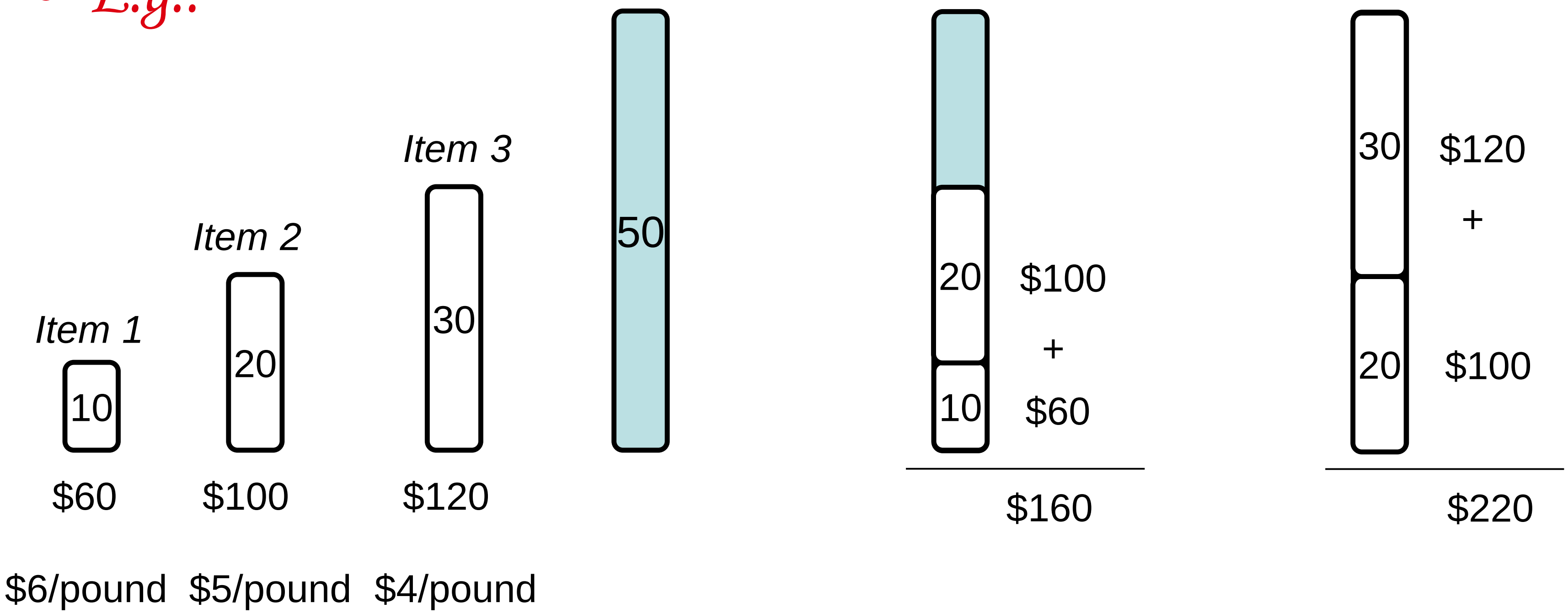
- Thief has a knapsack of capacity W
- There are n items: for i-th item value v_i and weight w_i
- Goal:
  - find x_i such that for all $x_i = \{0, 1\}, i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

0-1 Knapsack - Greedy Strategy

- *E.g.:*



- None of the solutions involving the greedy choice (item 1) leads to an optimal solution
  - The greedy choice property does not hold

0-1 Knapsack - Dynamic Programming

- $P(i, w)$ – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

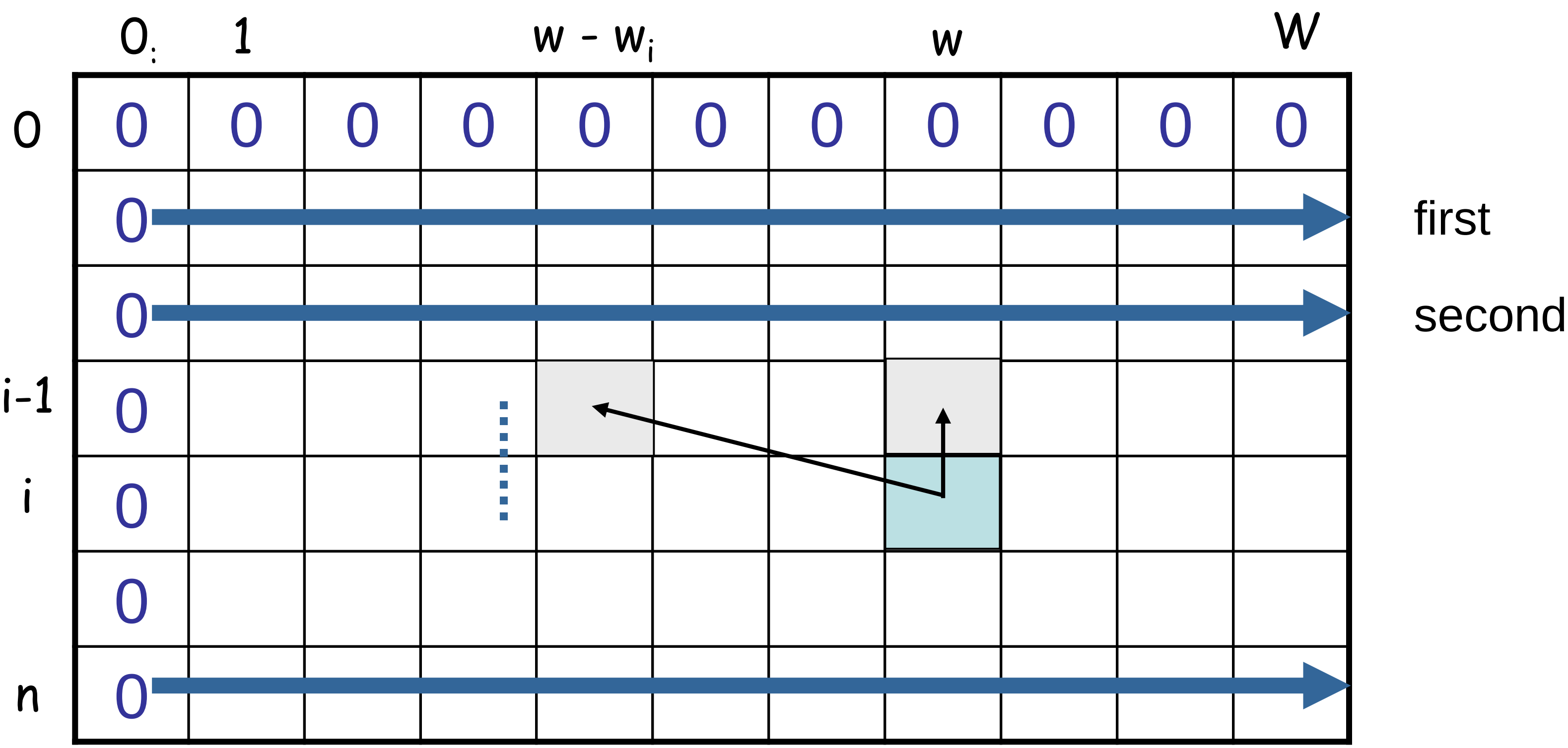
$$P(i, w) = v_i + P(i - 1, w - w_i)$$

- Case 2: thief does not take item i

$$P(i, w) = P(i - 1, w)$$

0-1 Knapsack - Dynamic Programming

$$P(i, w) = \max\{\overbrace{v_i + P(i-1, w-w_i)}^{\text{Item } i \text{ was taken}}, \overbrace{P(i-1, w)}^{\text{Item } i \text{ was not taken}}\}$$



Example:

$$P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}$$

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(4, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at $P(n, W)$
- When you go left-up $\Rightarrow$ item i has been taken
- When you go straight up $\Rightarrow$ item i has not been taken

Optimal Substructure

- Consider the most valuable load that weights at most W pounds
- If we remove item j from this load

$\Rightarrow$ The remaining load must be the most valuable load weighing at most $W - w_j$ that can be taken from the remaining $n - 1$ items

Overlapping Subproblems

$$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0										
i	0										
	0										
n	0										

E.g.: all the subproblems shown in grey may depend on $P(i-1, w)$