Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

Lecture 19

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 the hope of getting a globally optimal solution
- Greedy algorithms don’t always yield an optimal solution
- When the **problem** has certain general characteristics (**greedy choice property**), greedy algorithms give optimal solutions

Activity Selection

- Problem
  - Schedule the largest possible set of non-overlapping activities for a given room

	Start	End	Activity
1	8:00am	9:15am	Numerical methods class
2	8:30am	10:30am	Movie presentation (refreshments served)
3	9:20am	11:00am	Data structures class
4	10:00am	noon	Programming club mtg. (Pizza provided)
5	11:30am	1:00pm	Computer graphics class
6	1:05pm	2:15pm	Analysis of algorithms class
7	2:30pm	3:00pm	Computer security class
8	noon	4:00pm	Computer games contest (refreshments served)
9	4:00pm	5:30pm	Operating systems class

Activity Selection

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

  $S = \{a_1, . . . , a_n\}$ – 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
  - $0 \le s_i < f_i < \infty$
- Activities a_i and a_j are **compatible** if the intervals $[s_i , f_i)$ and $[s_j, f_j)$ do not overlap

$f_i \le s_j$ $\qquad\qquad$ $f_j \le s_i$

Activity Selection Problem

Select the largest possible set of non-overlapping (***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

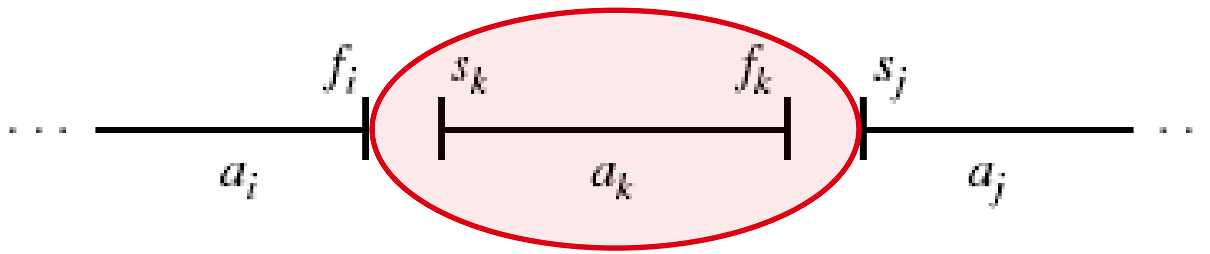
- Activities are sorted in increasing order of finish times
- A subset of mutually compatible activities: $\{a_3, a_9, a_{11}\}$
- Maximal set of mutually compatible activities: $\{a_1, a_4, a_8, a_{11}\}$ and $\{a_2, a_4, a_9, a_{11}\}$

Optimal Substructure

- Define the space of subproblems:

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

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



- 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

Representing the Problem

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

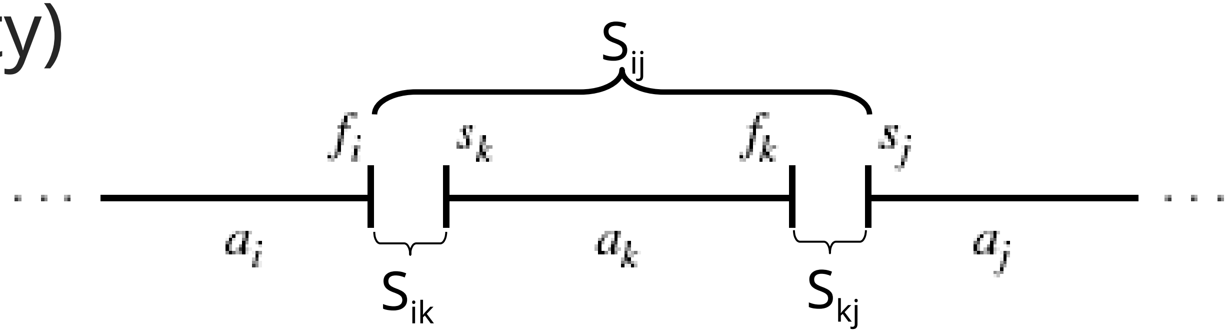
    contradiction with $f_i \geq f_j$!

  $\Rightarrow S_{ij} = \emptyset$ (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

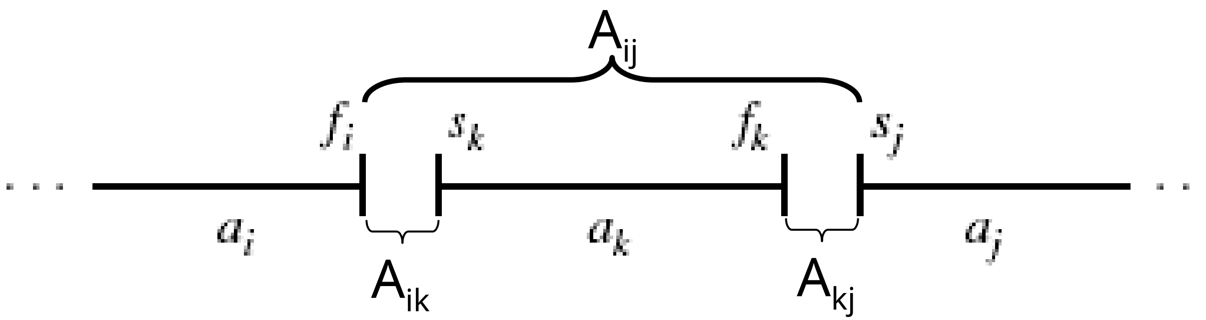
- Subproblem:
  - Select a maximum-size subset of mutually compatible activities from set S_{ij}
- Assume that a solution to the above subproblem includes activity a_k (S_{ij} is non-empty)



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

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

Optimal Substructure



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}

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

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

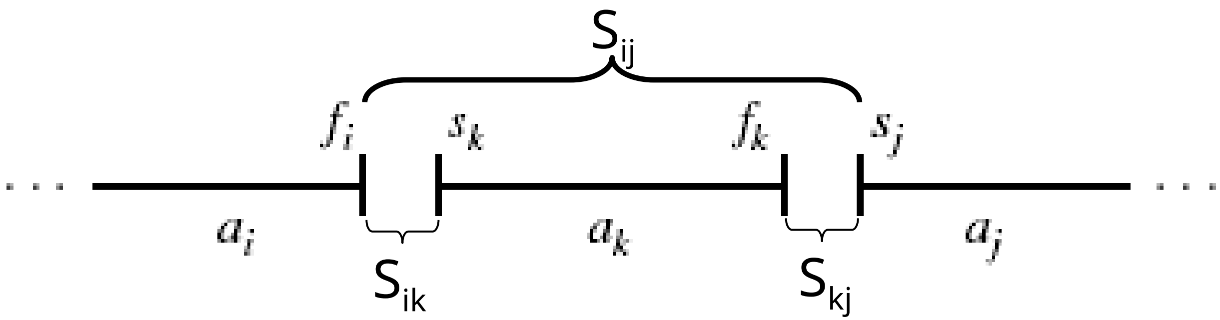
Recursive Solution

- 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

Recursive Solution



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

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

Recursive Solution

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

- 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})

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

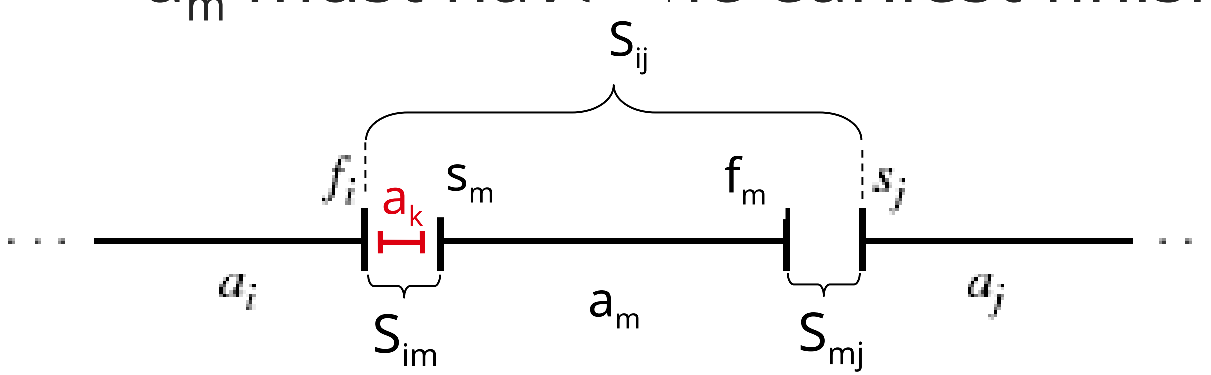
Proof

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

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

$\Rightarrow$ $f_k < f_m$ contradiction !

a_m must have the earliest finish time

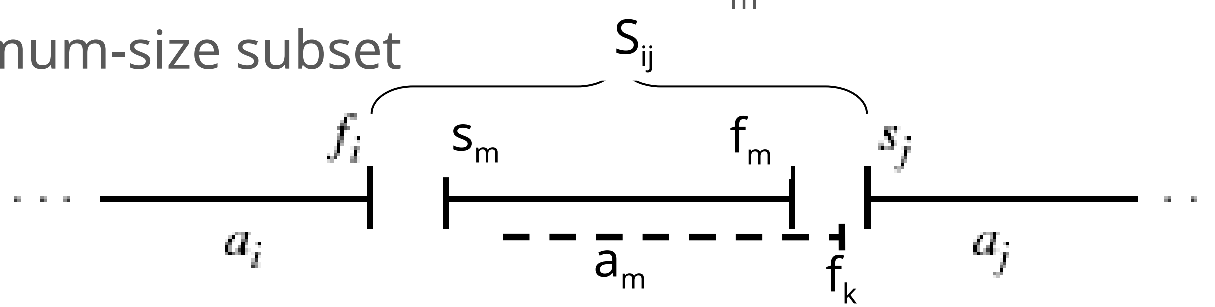


$\Rightarrow$ There is no $a_k \in S_{im} \Rightarrow S_{im} = \emptyset$

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 as $A_{ij} \Rightarrow a_m$ 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 a_m with the earliest finish time in S_{ij}

- Making the greedy choice (the activity with the earliest finish time in S_{ij})
  - Reduces the number of subproblems and choices
  - Allows solving each subproblem in a top-down fashion
- Only one subproblem left to solve!

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

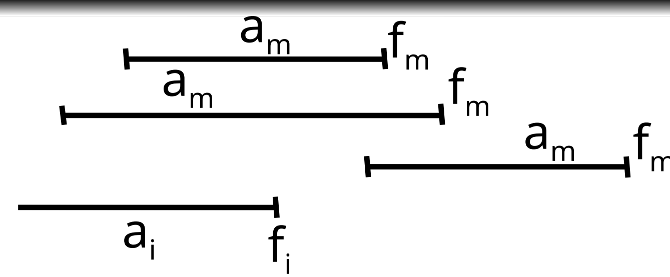
Characterizing the Subproblems

- 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



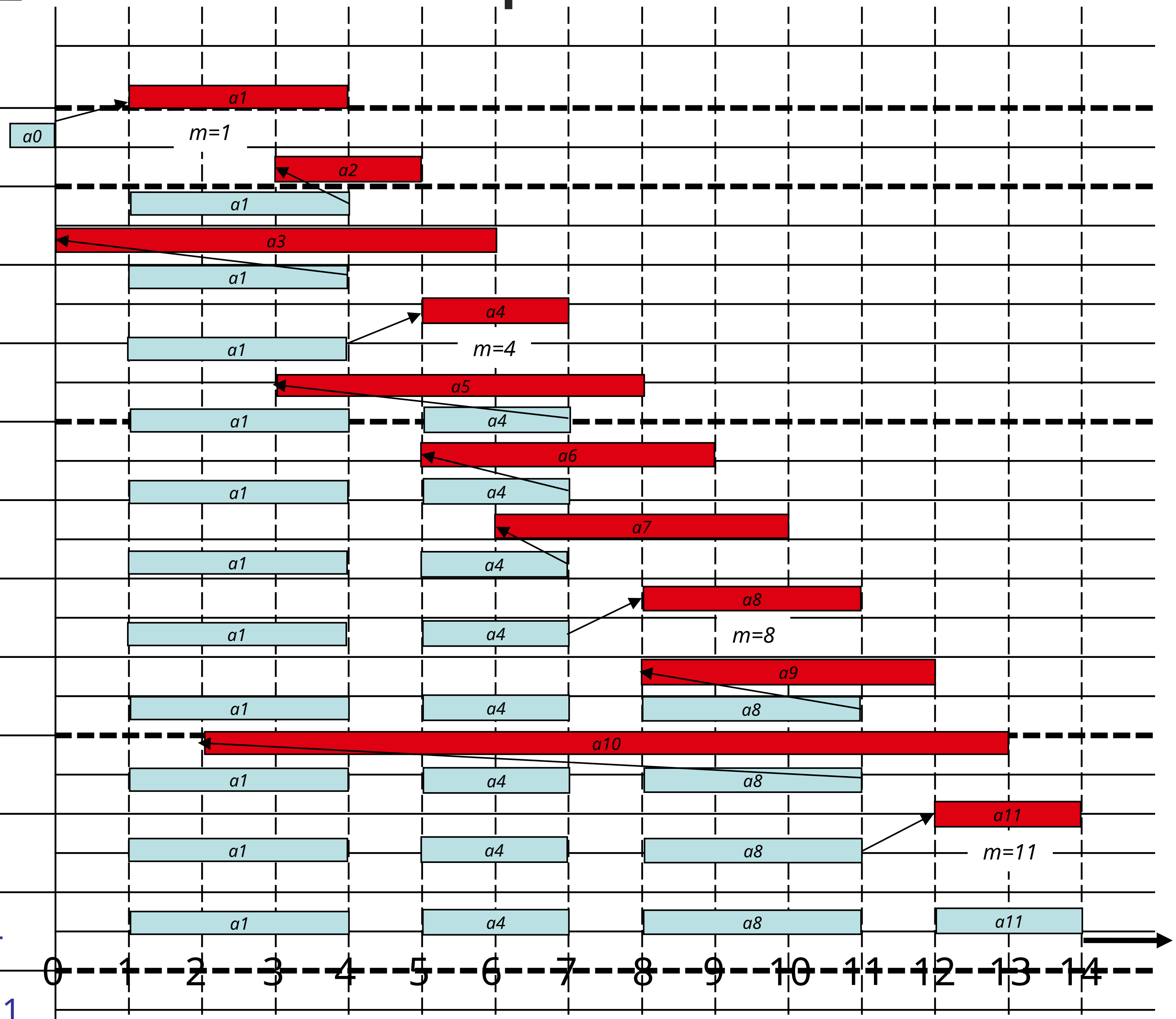
Alg.: REC-ACT-SEL (s, f, i, n)

1. $m \leftarrow i + 1$
2. **while** $m \leq n$ **and** $s_m < f_i$ ►Find first activity in $S_{i,n+1}$
3. **do** $m \leftarrow m + 1$
4. **if** $m \leq n$
5. **then return** $\{a_m\} \cup$ REC-ACT-SEL(s, f, m, n)
6. **else return** $\emptyset$

- 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)

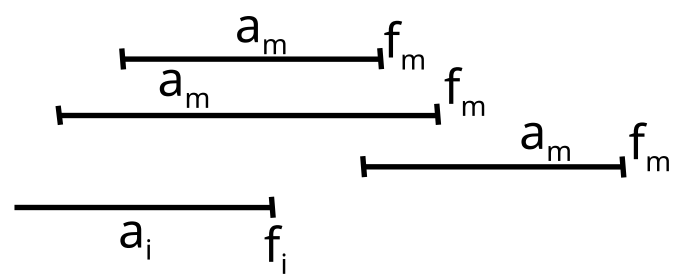
Example

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	∞	∞+1



An Incremental Algorithm

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




```
A ← {a_1}
i ← 1
for m ← 2 to n
    do if s_m ≥ f_i        ► activity a_m is compatible with a_i
        then A ← A ⋃ {a_m}
             i ← 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 (greedy) choice and are left with only one subproblem to solve
2. Prove the GREEDY CHOICE property:
   - that there is always an optimal solution to the original problem that makes the greedy choice
3. Prove the OPTIMAL SUBSTRUCTURE:
   - 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

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 robbing 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 fractions x_i so that for all $0 \le x_i \le 1$, $i = 1, 2, .., n$

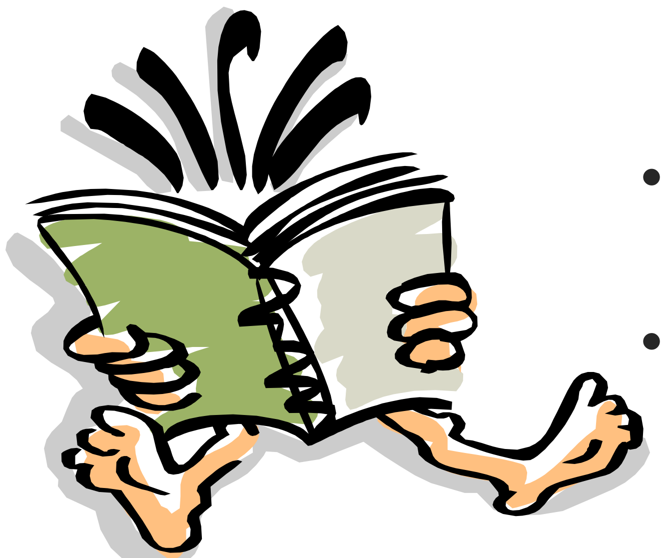
    $\sum w_i x_i \le W$ and

    $\sum x_i v_i$ 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 (choose item 1) = $v_1 W/w_1 = 2/100$
  - Smaller than what the thief can take if choosing the other item
    Total value (choose item 2) = $v_2 W/w_2 = 1$

Readings



- For this lecture
  - Chapter 15
- Coming next
  - Chapter 15