CE100 Algorithms and Programming II

Greedy Algorithms, Knapsack

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-7 (Greedy Algorithms, Knapsack)

#### Spring Semester, 2021-2022

Download [DOC](ce100-week-7-knapsack.tr.md_doc.pdf), [SLIDE](ce100-week-7-knapsack.tr.md_slide.pdf), [PPTX](ce100-week-7-knapsack.tr.md_slide.pptx)

## Greedy Algorithms, Knapsack

## Outline

* Greedy Algorithms and Dynamic Programming Differences
* Greedy Algorithms
  + Activity Selection Problem
  + Knapsack Problems
    - The 0-1 knapsack problem
    - The fractional knapsack problem

## **Activity Selection Problem**

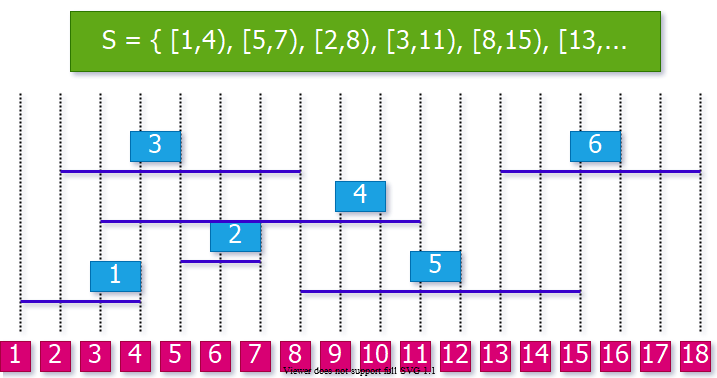
## Activity Selection Problem

* We have:
  + A set of activities with fixed start and finish times
  + One shared resource (only one activity can use at any time)
* **Objective:** Choose the max number of compatible activities
* **Note:** Objective is to maximize the number of activities, not the total time of activities.
* **Example:**
  + *Activities:* Meetings with fixed start and finish times
  + *Shared resource:* A meeting room
    - *Objective:* Schedule the max number of meetings

## Activity Selection Problem

* **Input:** a set of n activities
* : Start time of activity ,
* : Finish time of activity Activity takes place in
* **Aim:** Find max-size subset of mutually *compatible* activities
  + Max number of activities, not max time spent in activities
  + Activities and are compatible if intervals and do not overlap, i.e., either or

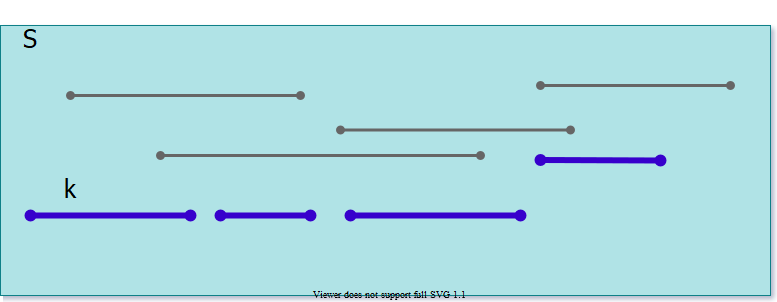
## Activity Selection Problem An Example



center h:400px

## Optimal Substructure Property

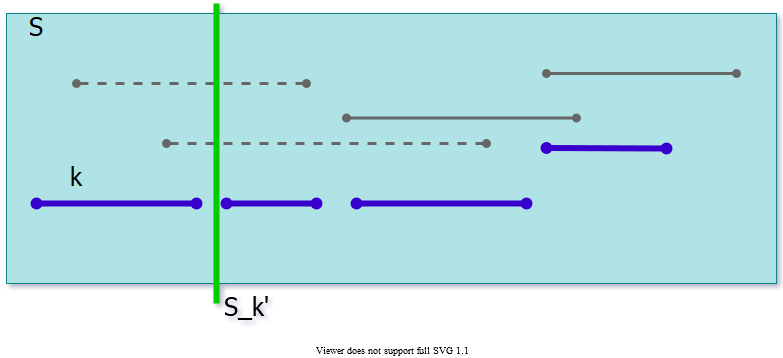
* Consider an optimal solution for activity set .
* Let be the activity in with the **earliest finish time**



center h:400px

## Optimal Substructure Property

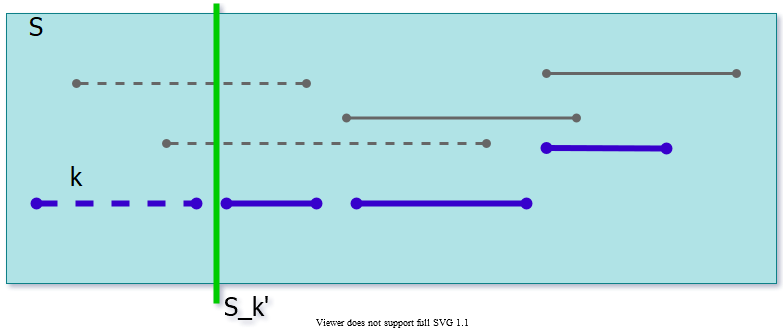
* Consider an optimal solution for activity set .
* Let be the activity in with the **earliest finish time**
* Now, consider the **subproblem**  that has the activities that start after finishes, i.e.
* What can we say about the optimal solution to ?



center h:300px

## Optimal Substructure Property

* Consider an optimal solution for activity set .
* Let be the activity in with the **earliest finish time**
* Now, consider the **subproblem**  that has the activities that start after finishes, i.e.
* is an optimal solution for . Why?



center h:300px

## Optimal Substructure

* **Theorem:** Let be the activity with the earliest finish time in an optimal soln then
  + is an optimal solution to subproblem
* **Proof (by contradiction):**
  + Let be an optimal solution to and
  + Then, is compatible and
  + Contradiction to the optimality of

## Optimal Substructure

* **Recursive formulation:** Choose the first activity , and then solve the remaining subproblem
* How to choose the first activity ?
  + DP, memoized recursion?
    - i.e. choose the value that will have the max size for
* DP would work,
  + but is it necessary to try all possible values for ?

## Greedy Choice Property

* Assume (without loss of generality)
  + If not, sort activities according to their finish times in non-decreasing order
* **Greedy choice property:** a sequence of locally optimal (greedy) choices an optimal solution
* How to choose the first activity **greedily** without losing optimality?

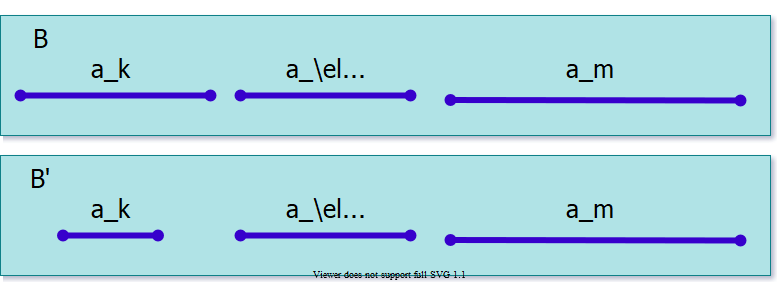
## Greedy Choice Property - Theorem

* Let activity set , where
* **Theorem:** There exists an optimal solution such that

**In other words, the activity with the earliest finish time is guaranteed to be in an optimal solution**.

## Greedy Choice Property - Proof

* **Theorem:** There exists an optimal solution such that
* **Proof:** Consider an arbitrary optimal solution , where
  + If , then starts with , and the proof is complete
  + If , then create another solution by replacing with . Since , is guaranteed to be valid, and , hence also optimal

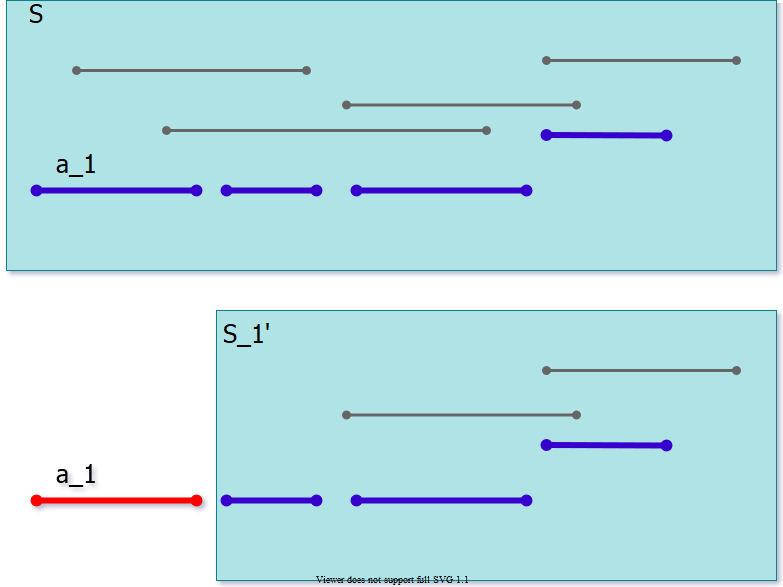


center h:250px

## Greedy Algorithm

* So far, we have:
  + **Optimal substructure property:** If is an optimal solution, then must be optimal for subproblem , where
    - **Note:** is the activity with the earliest finish time in
  + **Greedy choice property:** There is an optimal solution that contains
    - **Note:** is the activity with the earliest finish time in

## Greedy Algorithm



center h:450px

*explained in the next slide..*

## Greedy Algorithm

* **Theorem:** There exists an optimal solution such that
* Basic idea of the greedy algorithm:
  + Add to
  + Solve the remaining subproblem , and then append the result to
* **Remember arbitary optimal solution explaination from previous sections (finish time order is important for selection with star time and overlapping checking)**
  + ,
  + where

## Greedy Algorithm for Activity Selection

### Definitions in Greedy Algorithm:

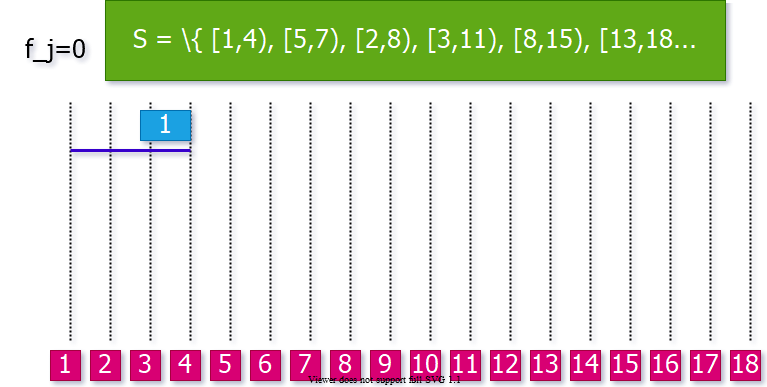
* : specifies the index of most recent activity added to
* , max finish time of any activity in ;
  + because activities are processed in non-decreasing order of finish times
* Thus, checks the compatibility of to current
* **Running time:** assuming that the activities were already sorted.

## Greedy Algorithm for Activity Selection

### Pseudocode for Greedy Algorithm:

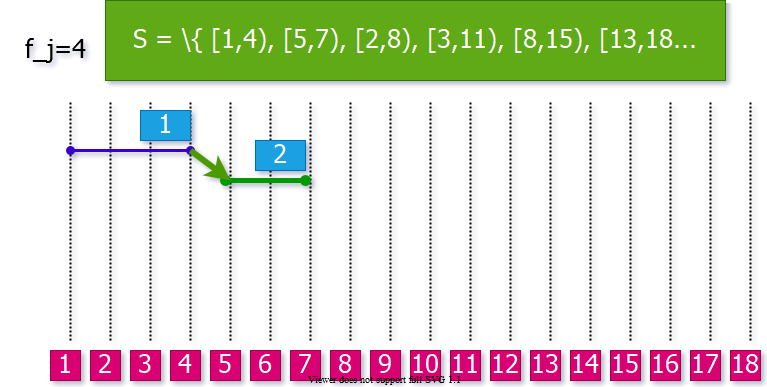
$$ $$

## Greedy Algorithm for Activity Selection, An Example (Step-1)



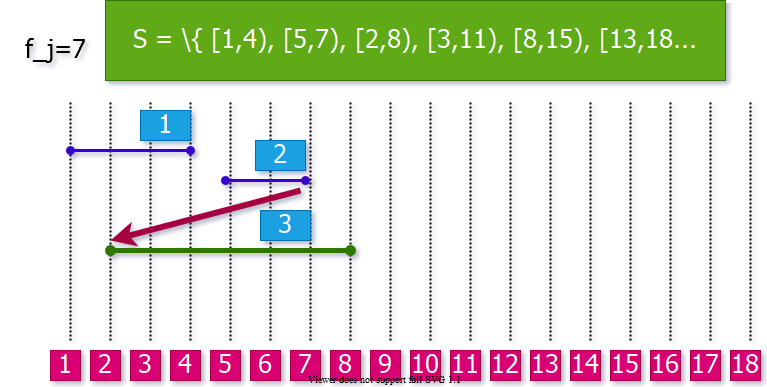
center h:450px

## Greedy Algorithm for Activity Selection, An Example (Step-2)



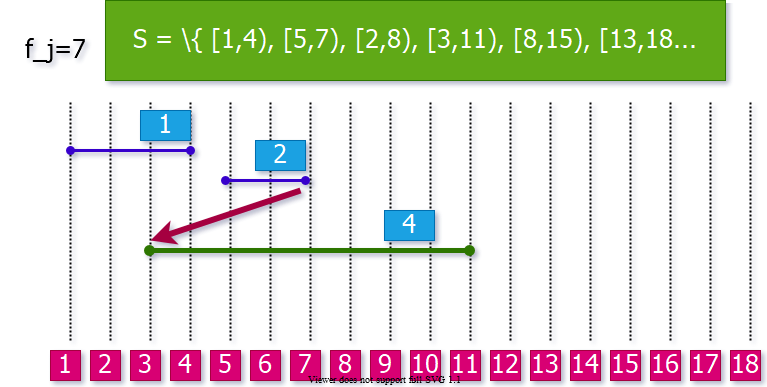
center h:450px

## Greedy Algorithm for Activity Selection, An Example (Step-3)



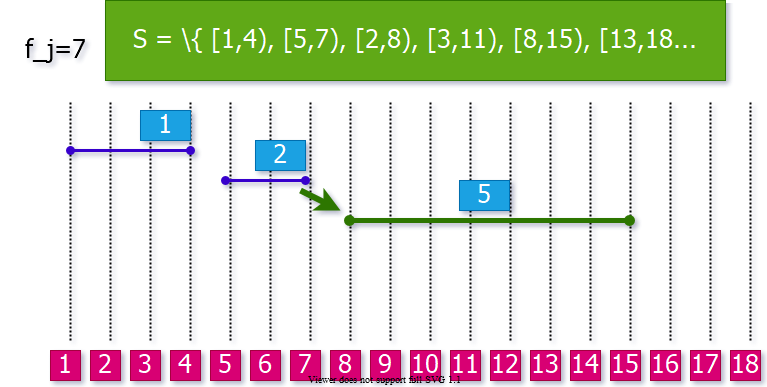
center h:450px

## Greedy Algorithm for Activity Selection, An Example (Step-4)



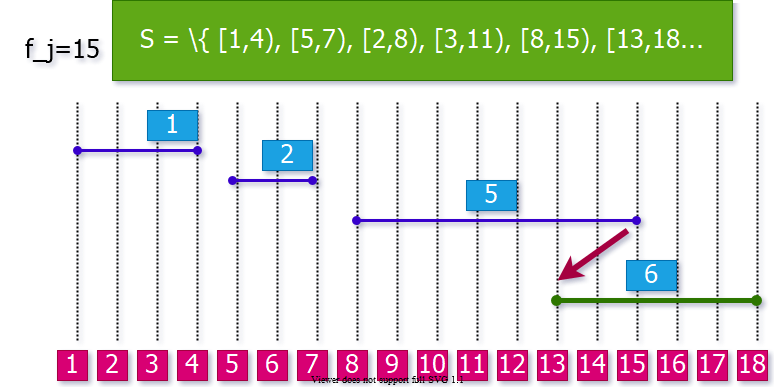
center h:450px

## Greedy Algorithm for Activity Selection, An Example (Step-5)



center h:450px

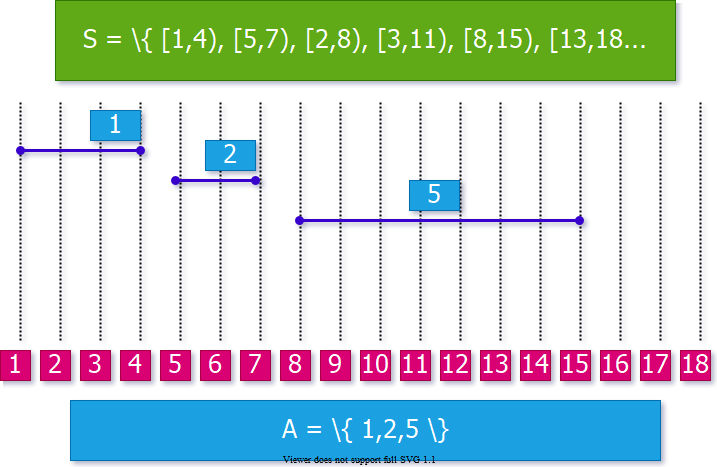
## Greedy Algorithm for Activity Selection, An Example (Step-6)



center h:450px

## Greedy Algorithm for Activity Selection, An Example (Step-7)

### Final Solution



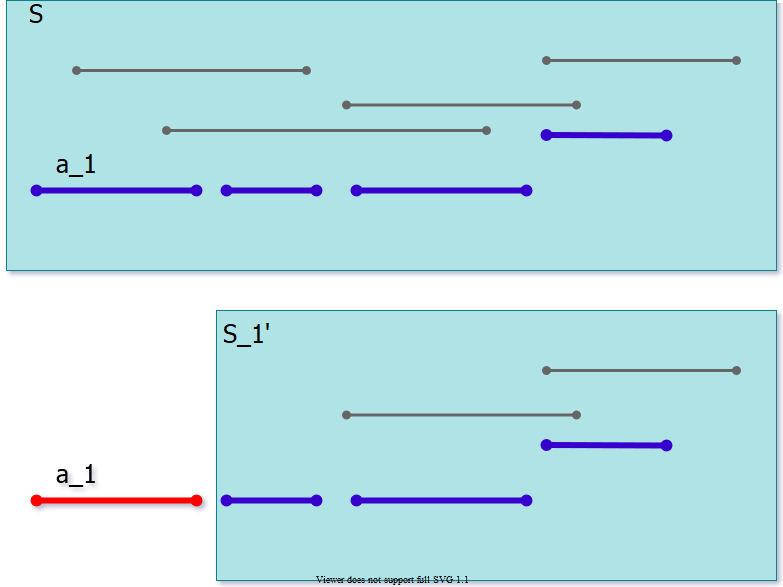
center h:450px

## **Comparison of DP and Greedy Algorithms**

## **Reminder**: DP-Based Matrix Chain Order

* We don’t know ahead of time which value to choose.
* We first need to compute the results of subproblems and before computing
* The selection of is done based on the **results of the subproblems**.

## Greedy Algorithm for Activity Selection



center h:450px

*explained in the next slide..*

## Greedy Algorithm for Activity Selection

* Make a greedy selection in the beginning:
  + Choose (the activity with the earliest finish time)
* Solve the remaining subproblem (all activities that start after a1)

## Greedy vs Dynamic Programming

* Optimal substructure property exploited by both **Greedy** and **DP** strategies
* **Greedy Choice Property:** A sequence of locally optimal choices an optimal solution
  + We make the choice that seems best at the moment
  + Then solve the subproblem arising after the choice is made
* **DP:** We also make a choice/decision at each step, but the choice may depend on the optimal solutions to subproblems
* **Greedy:** The choice may depend on the choices made so far, but it cannot depend on any future choices or on the solutions to subproblems

## Greedy vs Dynamic Programming

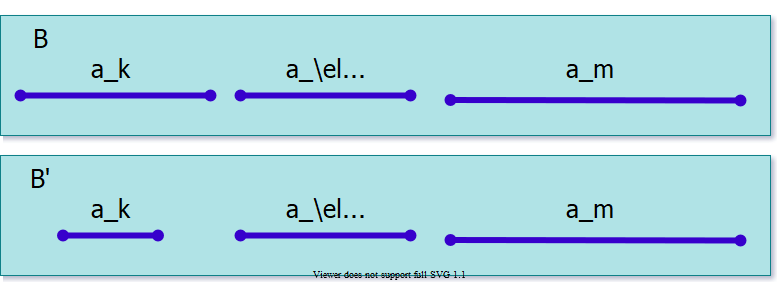
* **DP** is a bottom-up strategy (*use memory to store the results of subproblems*)
* **Greedy** is a top-down strategy (*make choices at each step*)
  + each greedy choice in the sequence iteratively reduces each problem to a similar but smaller problem

## Proof of Correctness of Greedy Algorithms

* Examine a globally optimal solution
* Show that this soln can be modified so that
  + 1. A greedy choice is made as the first step
    2. This choice reduces the problem to a similar but smaller problem
* Apply induction to show that a greedy choice can be used at every step
* Showing (2) reduces the proof of correctness to proving that the problem exhibits optimal substructure property

## Greedy Choice Property - Proof

* **Theorem:** There exists an optimal solution such that
* **Proof:** Consider an arbitrary optimal solution , where
  + If , then starts with , and the proof is complete
  + If , then create another solution by replacing with . Since , is guaranteed to be valid, and , hence also optimal



center h:250px

## Elements of Greedy Strategy

* How can you judge whether
* A greedy algorithm will solve a particular optimization problem?
* **Two key ingredients**
  + Greedy choice property
  + Optimal substructure property

## Key Ingredients of Greedy Strategy

* **Greedy Choice Property:** A globally optimal solution can be arrived at by making locally optimal (greedy) choices
* In **DP**,we make a choice at each step but the choice may depend on the solutions to subproblems
* In **Greedy Algorithms**, we make the choice that seems best at that moment then solve the subproblems arising after the choice is made
  + The choice may depend on choices so far, but it cannot depend on any future choice or on the solutions to subproblems
* DP solves the problem bottom-up
* Greedy usually progresses in a top-down fashion by making one greedy choice after another reducing each given problem instance to a smaller one

## Key Ingredients: **Greedy Choice Property**

* We must prove that a greedy choice at each step yields a globally optimal solution
* The proof examines a globally optimal solution
* Shows that the soln can be modified so that a **greedy choice made as the first step** reduces the problem to a similar but smaller subproblem
* Then **induction** is applied to show that a greedy choice can be used at each step
* Hence, this induction proof reduces the proof of correctness to demonstrating that an optimal solution must exhibit **optimal substructure** property

## Key Ingredients: **Greedy Choice Property**

* **How to prove the greedy choice property?**
  + Consider the greedy choice
  + Assume that there is an **optimal solution that doesn’t contain** .
  + Show that it is possible to **convert**  to another optimal solution , where contains .
* **Example:** Activity selection algorithm
  + Greedy choice: (the activity with the earliest finish time)
  + Consider an optimal solution without
  + Replace the first activity in with to construct
  + Prove that must be an optimal solution

## Key Ingredients: **Optimal Substructure**

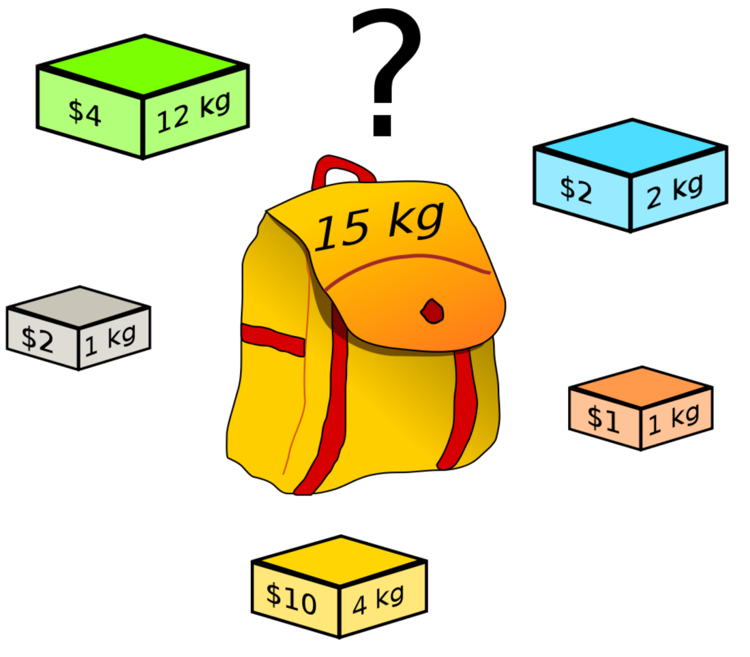
* A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems
* **Example:** Activity selection problem
  + If an optimal solution A to S begins with activity a1 then the set of activities
  + is an optimal solution to the activity selection problem
  + where is the start time of activity and is the finish time of activity

## Key Ingredients: **Optimal Substructure**

* Optimal substructure property is exploited by both Greedy and dynamic programming strategies
* Hence one may
  + Try to generate a dynamic programming soln to a problem when a greedy strategy suffices  inefficient
  + Or, may mistakenly think that a greedy soln works when in fact a DP soln is required  incorrect
* **Example:** Knapsack Problems(S, w)

## **Knapsack Problems**

## Knapsack Problem



bg right:40% h:400px

* Each item has:
  + weight
  + value
* A thief has a knapsack of weight capacity
* **Which items to choose to maximize the value of the items in the knapsack?**

## Knapsack Problem: **Two Versions**

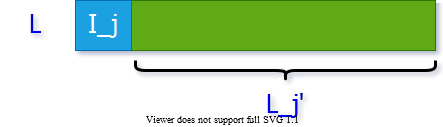
* **The 0-1 knapsack problem:**
  + Each item is discrete.
  + Each item either chosen as a whole or not chosen.
  + **Examples:** *TV, laptop, gold bricks, etc*.
* **The fractional knapsack problem:**
  + Can choose fractional part of each item.
  + If item i has weight wi, we can choose any amount ≤ wi
  + **Examples:** *Gold dust, silver dust, rice, etc.*

## Knapsack Problems

* **The 0-1 Knapsack Problem()**
  + A thief robbing a store finds items , the ith item is worth dollars and weighs pounds, where vi and wi are integers
  + He wants to take as valuable a load as possible, but he can carry at most pounds in his knapsack, where is an integer
  + The thief cannot take a fractional amount of an item
* **The Fractional Knapsack Problem ()**
  + The scenario is the same
  + But, the thief can take fractions of items rather than having to make binary () choice for each item

## Optimal Substructure Property for the 0-1 Knapsack Problem (S, W)

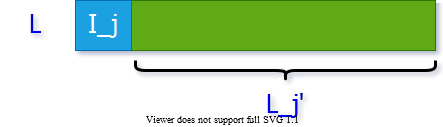
* Consider an optimal load L for the problem (S, W).
* Let Ij be an item chosen in L with weight wj
* Assume we remove Ij from L, and let:
* Q: *What can we say about the optimal substructure property?*



center h:150px

## Optimal Substructure Property for the 0-1 Knapsack Problem (S, W)

* Optimal substructure property:
  + must be an optimal solution for
* **Why?**
  + If we remove item from , we can construct a new optimal solution for
  + If is optimal, then must be optimal



center h:150px

## Optimal Substructure Property for the 0-1 Knapsack Problem (S, W)

* Optimal substructure: must be an optimal solution for
* **Proof:** By contradiction, assume there is a solution for , which is better than .
  + We can construct a solution B for the original problem () as: .
  + The total value of is now higher than , which is a contradiction because is optimal for .

## Optimal Substructure Property for the Fractional Knapsack Problem (S, W)

* Consider an optimal solution L for (S, W)
* If we remove a weight of item from optimal load and let:
  + The remaining load
  + must be a most valuable load weighing at most
  + pounds that the thief can take from
* That is, Lj´ should be an optimal soln to the

## Knapsack Problems

* Two different problems:
  + 0-1 knapsack problem
  + Fractional knapsack problem
* The problems are similar.
  + Both problems have optimal substructure property.
* Which algorithm to solve each problem?

## Fractional Knapsack Problem

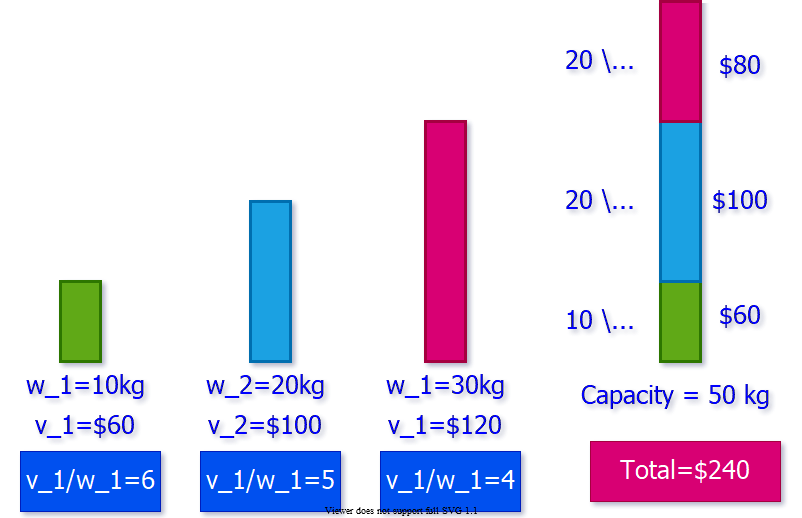
* Can we use a greedy algorithm?
* Greedy choice: Take as much as possible from the item with the largest value per pound
* Does the greedy choice property hold?
  + Let be the item with the largest value per pound
  + Need to prove that there is an optimal load that has as much as possible.
  + **Proof:** *Consider an optimal solution L that does not have the maximum amount of item . We could replace the items in with item until has maximum amount of . would still be optimal, because item has the highest value per pound.*

## Greedy Solution to Fractional Knapsack

* 1. Compute the value per pound for each item
  2. The thief begins by taking, as much as possible, of the item with the greatest value per pound
  3. If the supply of that item is exhausted before filling the knapsack, then he takes, as much as possible, of the item with the next greatest value per pound
  4. Repeat (2-3) until his knapsack becomes full

**Thus, by sorting the items by value per pound the greedy algorithm runs in time**

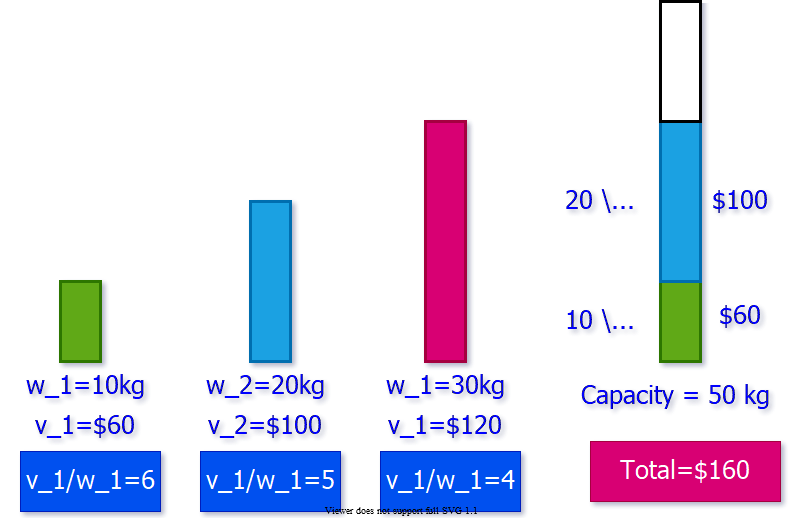
## Fractional Knapsack Problem: **Example**



center h:450px

## 0-1 Knapsack Problem

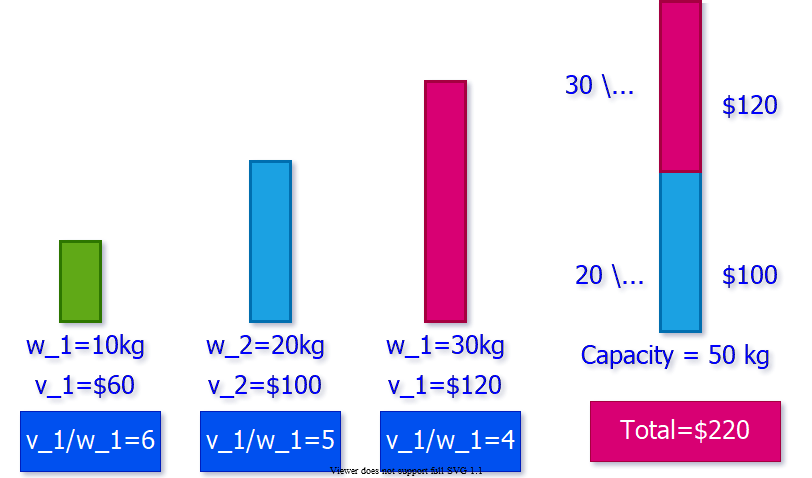
* Can we use the same greedy algorithm?
  + Is there a better solution?



center h:450px

## 0-1 Knapsack Problem

* The optimal solution for this problem is:
  + This solution cannot be obtained using the greedy algorithm



center h:450px

## 0-1 Knapsack Problem

* When we consider an item for inclusion we must compare the solutions to two subproblems
  + Subproblems in which is included and excluded
* The problem formulated in this way gives rise to many
  + **overlapping subproblems** (a key ingredient of DP)
    - In fact, dynamic programming can be used to solve the **0-1 Knapsack problem**

## 0-1 Knapsack Problem

* A thief robbing a store containing articles
* The value of article is dollars ( is integer)
* The weight of article is kg ( is integer)
* Thief can carry at most kg in his knapsack
* Which articles should he take to maximize the value of his load?
* Let denote 0-1 knapsack problem
* Consider the solution as a sequence of decisions
  + **i.e.**, decision: whether thief should pick for optimal load.

## Optimal Substructure Property

* Notation: :
  + The items to choose from:
  + The knapsack capacity:
* Consider an optimal load for problem
* Let’s consider two cases:
  + is **in**
  + is **not in**

## Optimal Substructure Property

* **Case 1:** If
  + What can we say about the optimal substructure?
    - must be optimal for
    - :
      * The items to choose from
      * The knapsack capacity:
* **Case 2:** If
  + What can we say about the optimal substructure?
    - must be optimal for
      * :
      * The items to choose from
      * The knapsack capacity:

## Optimal Substructure Property

* In other words, optimal solution to contains an optimal solution to:
  + either: (if is selected)
  + or: (if is not selected)

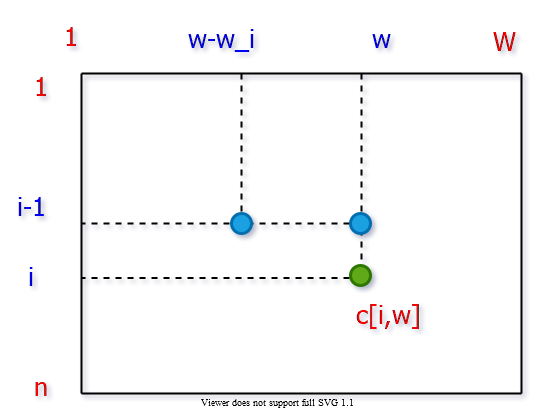
## Recursive Formulation

## 0-1 Knapsack Problem

* **Recursive definition for value of optimal soln:**
  + This recurrence says that an optimal solution for
    - either contains
    - or does not contain
  + If thief decides to pick
    - He takes value and he can choose from up to the weight limit to get
  + If he decides not to pick
    - He can choose from up to the weight limit to get
  + The better of these two choices should be made

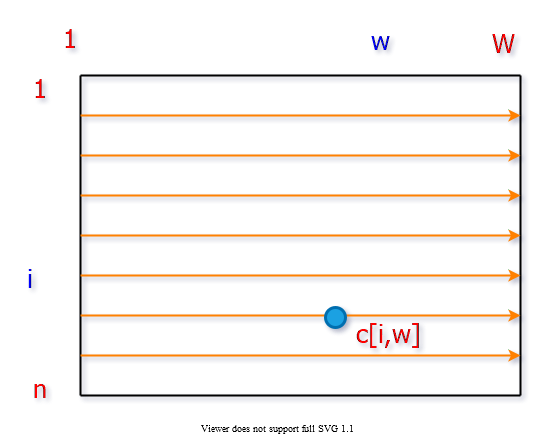
## Bottom-up Computation

* Need to process:
* after computing:
  + ,
    - for all



bg right:50% h:500px

## Bottom-up Computation



bg right:50% h:500px

## DP Solution to 0-1 Knapsack

* is an array;
* **Note** : table is computed in row-major order
* **Run time:**

## DP Solution to 0-1 Knapsack

$$ $$

## Constructing an Optimal Solution

* No extra data structure is maintained to keep track of the choices made to compute
  + i.e. The choice of whether choosing item i or not
* Possible to understand the choice done by comparing with
  + If then it means item i was assumed to be not chosen for the best

## Finding the Set S of Articles in an Optimal Load

$$ $$

## References

* [Introduction to Algorithms, Third Edition | The MIT Press](https://mitpress.mit.edu/books/introduction-algorithms-third-edition)
* [Bilkent CS473 Course Notes (new)](http://nabil.abubaker.bilkent.edu.tr/473/)
* [Bilkent CS473 Course Notes (old)](http://cs.bilkent.edu.tr/~ugur/teaching/cs473/)