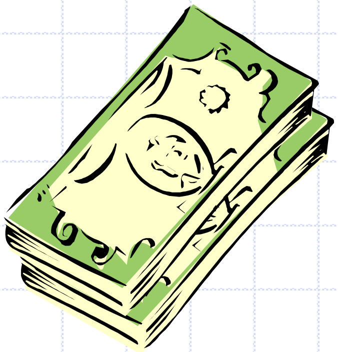
The Greedy Method



Outline and Reading

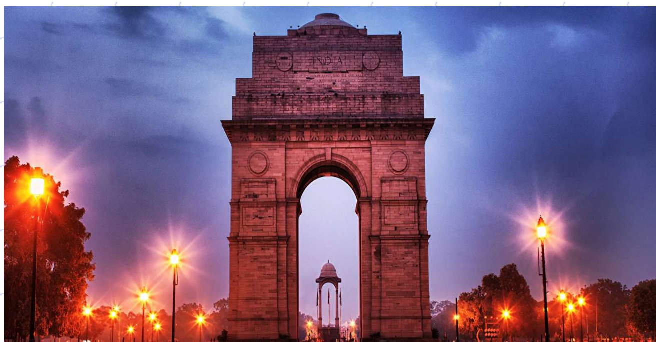
- The Greedy Method Technique
- Fractional Knapsack Problem
- Task Scheduling
- Minimum Spanning Trees

Small Thinking

Small Thinking

Greedy!

Problem: Travel



1,500 KM

Walk

CAB

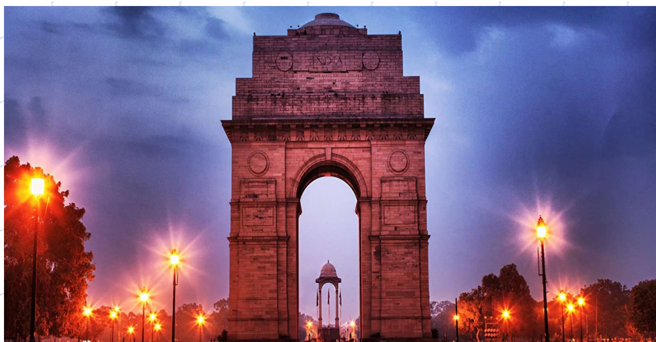
Public Road Transport

Train

Flight

Greedy!

Problem: Travel



1,500 KM

24 Hrs

Walk

CAB

Public Road Transport

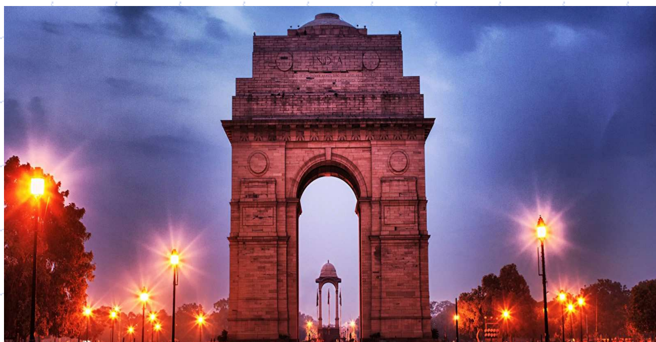
Train

Flight

Feasible Solution:
Train & Flight

Greedy!

Problem: Travel



1,500 KM

24 Hrs

< 2000 INR

Walk

CAB

Public Road Transport

Train

Flight

Feasible Solution: Train & Flight

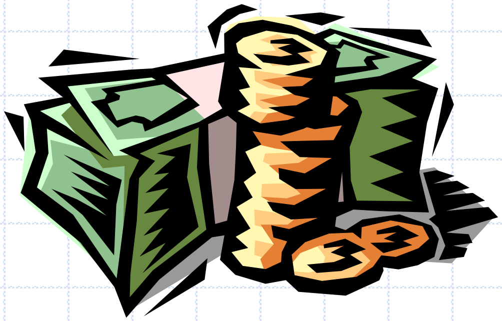
Optimal Solution: Train

Optimization Problem

The Greedy Method Technique

- **The greedy method** is a general algorithm design paradigm, built on the following elements:
  - **configurations**: different choices, collections, or values to find
  - **objective function**: a score assigned to configurations, which we want to either maximize or minimize
- It works best when applied to problems with the **greedy-choice** property:
  - a globally-optimal solution can always be found by a series of local improvements from a starting configuration.

Making Change



- Problem: A dollar amount to reach and a collection of coin amounts to use to get there.
- Configuration: A dollar amount yet to return to a customer plus the coins already returned
- Objective function: Minimize number of coins returned.
- Greedy solution: Always return the largest coin you can
- Example 1: Coins are valued \$.32, \$.08, \$.01
  - Has the greedy-choice property, since no amount over \$.32 can be made with a minimum number of coins by omitting a \$.32 coin (similarly for amounts over \$.08, but under \$.32).
- Example 2: Coins are valued \$.30, \$.20, \$.05, \$.01
  - Does not have greedy-choice property, since \$.40 is best made with two \$.20's, but the greedy solution will pick three coins (which ones?)

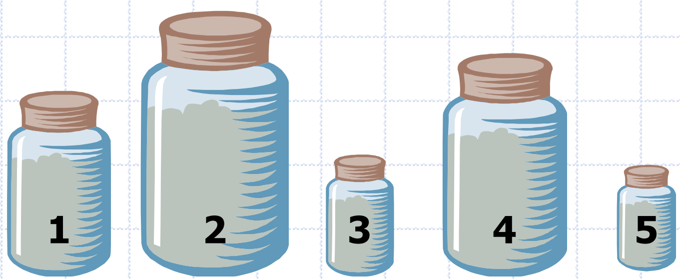
The Fractional Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize $\sum_{i \in S} b_i (x_i / w_i)$
  - Constraint: $\sum_{i \in S} x_i \le W$

Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50
Value: ($ per ml)	3	4	20	5	50

"knapsack"

10 ml

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

The Fractional Knapsack Algorithm

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $\sum_{i \in S} b_i(x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Run time: O(n log n). Why?
- Correctness: Suppose there is a better solution
  - there is an item i with higher value than a chosen item j, but $x_i < w_i$, $x_j > 0$ and $v_i < v_j$
  - If we substitute some i with j, we get a better solution
  - How much of i: $\min\{w_i - x_i, x_j\}$
  - Thus, there is no better solution than the greedy one

Algorithm *fractionalKnapsack*(*S*, *W*)

Input: set S of items w/ benefit b_i and weight w_i; max. weight W

Output: amount x_i of each item i to maximize benefit w/ weight at most W

for *each item i in S*

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

$w \leftarrow 0$ {total weight}

while $w < W$

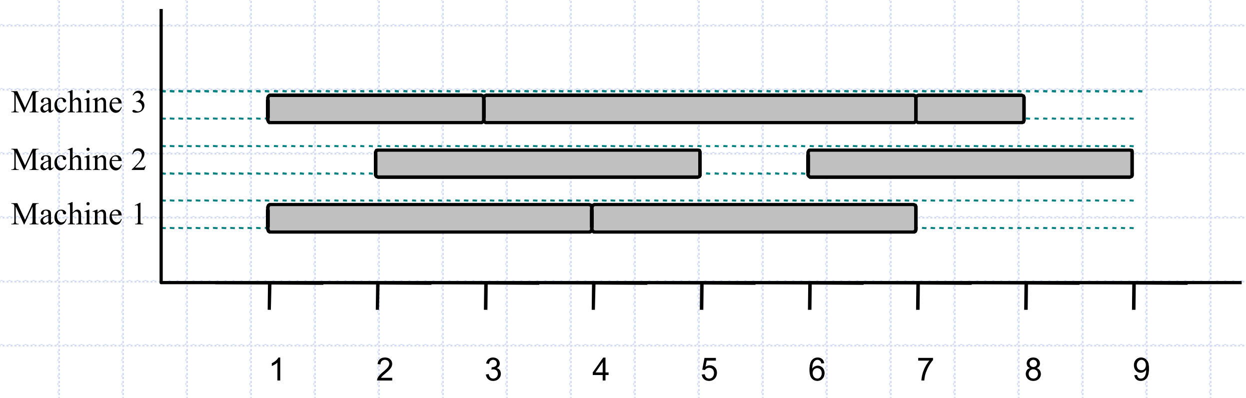
remove item i w/ highest v_i

$x_i \leftarrow \min\{w_i, W - w\}$

$w \leftarrow w + \min\{w_i, W - w\}$

Task Scheduling

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
- Goal: Perform all the tasks using a minimum number of "machines."



Task Scheduling Algorithm

- Greedy choice: consider tasks by their start time and use as few machines as possible with this order.
  - Run time: O(n log n). Why?
- Correctness: Suppose there is a better schedule.
  - We can use k-1 machines
  - The algorithm uses k
  - Let i be first task scheduled on machine k
  - Machine i must conflict with k-1 other tasks
  - But that means there is no non-conflicting schedule using k-1 machines

Algorithm ***taskSchedule(T)***

Input: set T of tasks w/ start time s_i and finish time f_i

Output: non-conflicting schedule with minimum number of machines

```
m ← 0                    {no. of machines}
while T is not empty
    remove task i w/ smallest s_i
    if there's a machine j for i then
        schedule i on machine j
    else
        m ← m + 1
        schedule i on machine m
```

Example

- Given: a set T of n tasks, each having:
  - A start time, s_i
  - A finish time, f_i (where $s_i < f_i$)
  - [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordered by start)
- Goal: Perform all tasks on min. number of machines