Chapter 12: Greedy Algorithms

Dr. Sirasit Lochanachit

Outline

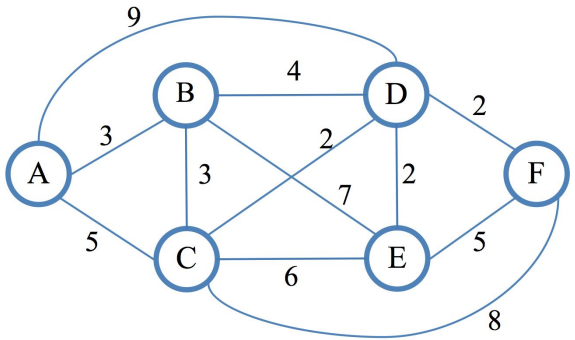
Optimisation Problems and Greedy Approach

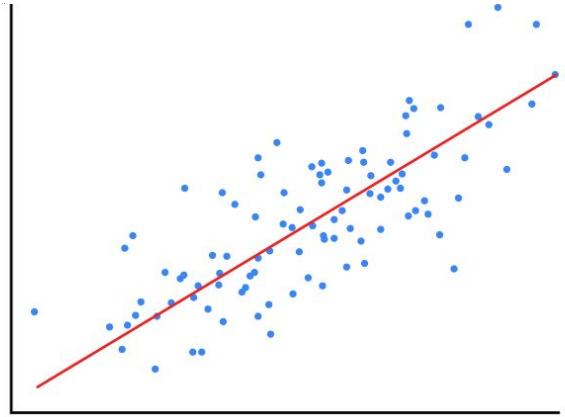
- Making change using fewest coins problem
- Knapsack Problems

Introduction

- In an algorithm design, there is no one '**silver bullet**' that is a cure for all computation problems.
- Different problems require the use of different kinds of techniques.
- A good programmer uses all these techniques based on the type of problem.

Optimisation





- Many problems involve optimisation of some value:
  - Find the shortest path between 2 points
  - Find the line that best fits a set of points
  - Find the smallest set of objects that satisfies some criteria
- Many strategies to solve these optimisation problems.
  - **Greedy Method**
  - **Dynamic Programming**

Optimisation Problem - Coin Exchange

Suppose you are a programmer for a vending machine manufacturer.

- Your company wants to streamline effort by giving out the fewest possible coins in change for each transaction.
- For example, a customer puts in a 50 Baht bill to buy an item valued at 37 Baht.

What is the **smallest number of coins** to make change?

Ref: http://www.thaismescenter.com/wp-content/uploads/2017/02/er10.jpg

Greedy Algorithm

- A greedy algorithm, as the name suggests, always **makes the choice that seems to be the best at that moment**.
- This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
    - Keep in mind that local optimal is not always global optimal.

Local Optimum

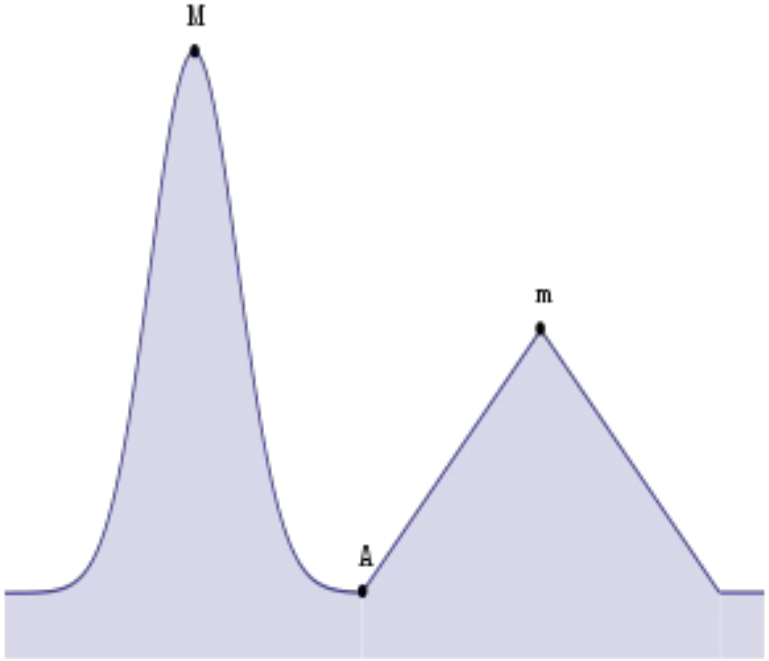
How do you decide which choice is optimal?

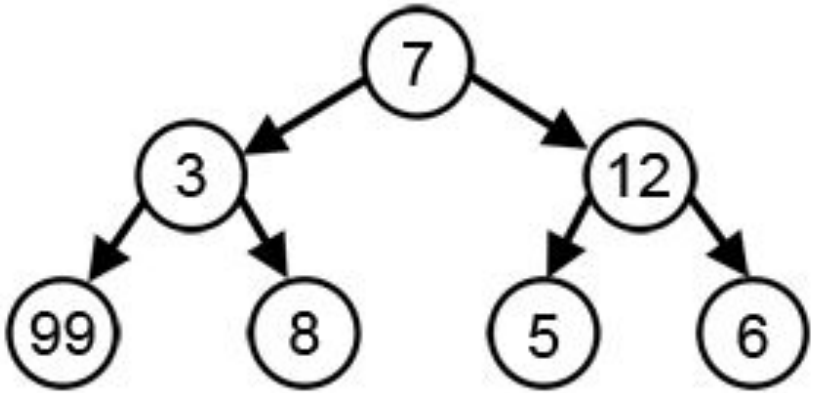
- Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point.
- A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized.
- The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

Local Optimum

- In many problems, a greedy strategy **does not usually produce an optimal solution**, but nonetheless a greedy heuristic may yield locally optimal solutions that **approximate** a globally optimal solution in a reasonable amount of time.
- Imagine there are 10, 7, 5, 2, and 1 Baht coins:
  - A change for 14 Baht is needed.
  - Greedy method would find the solution to be 10 + 2 + 2, which are 3 coins.
    - Is this the optimal solution?

Greedy Method - find Max





Making Change using Fewest Coins

A recursive solution is possible.

- Base case: If change is the same amount as the value of one of the coins, the answer is one coin.
- Recursive case: If the amount does not match, find the minimum of a coin as follows

$$\text{numCoins} = \min \begin{cases} 1 + \text{numCoins}(\text{originalAmount} - 1) \\ 1 + \text{numCoins}(\text{originalAmount} - 2) \\ 1 + \text{numCoins}(\text{originalAmount} - 5) \\ 1 + \text{numCoins}(\text{originalAmount} - 10) \end{cases}$$

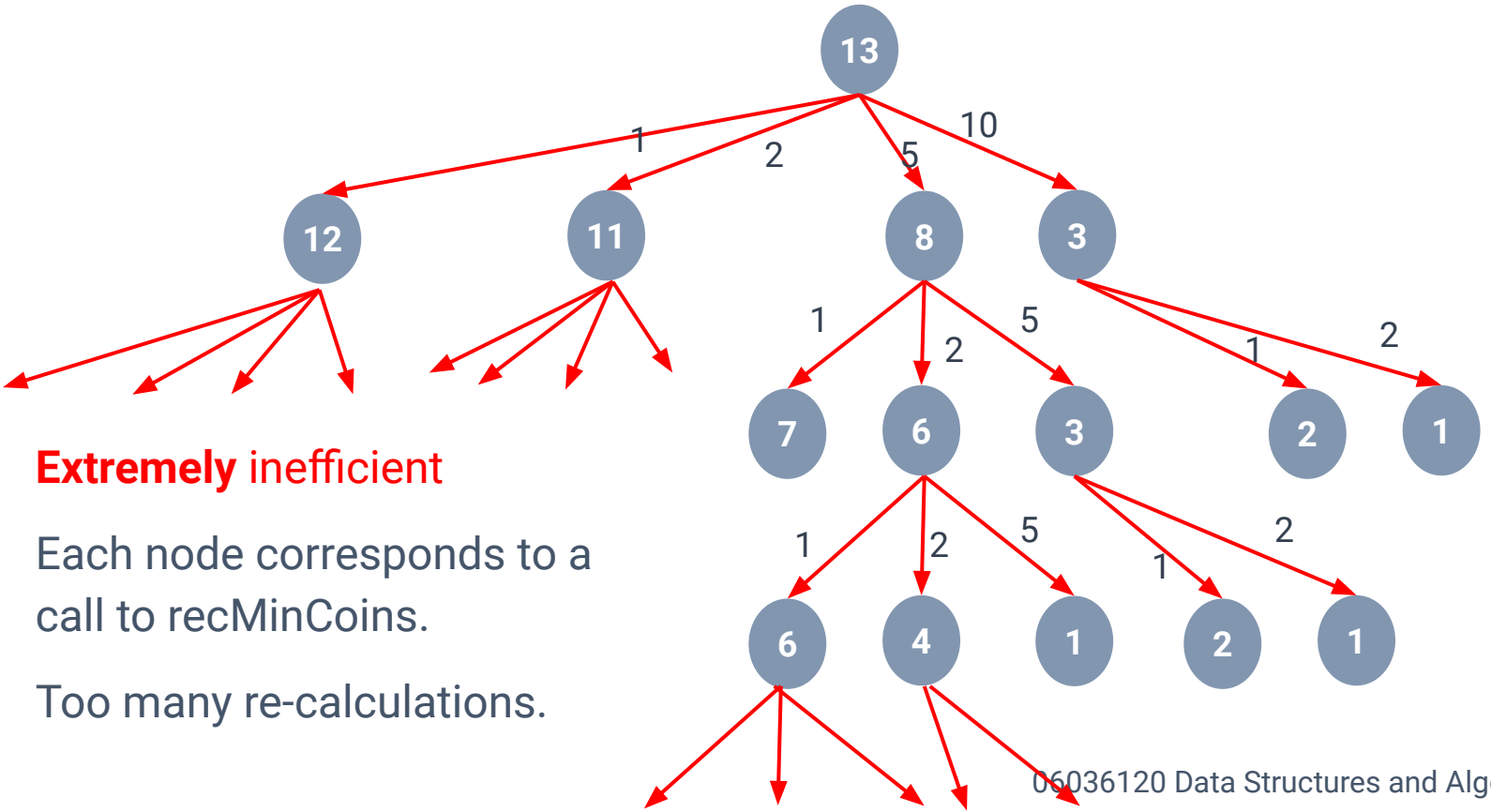
Making Change using Fewest Coins

Algorithm

```
def recMinCoin(coinValueList, change):
    minCoins = change
    if change in coinValueList:              # Base Case
        return 1
    else:                                    # Recursive Case
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMinCoin(coinValueList, change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins
print(recMinCoin([1, 2, 5, 10], 13))
```

Efficient?

Making Change using Fewest Coins



Extremely inefficient

Each node corresponds to a call to recMinCoins.

Too many re-calculations.

Making Change using Fewest Coins

Algorithm with Table (Caching)

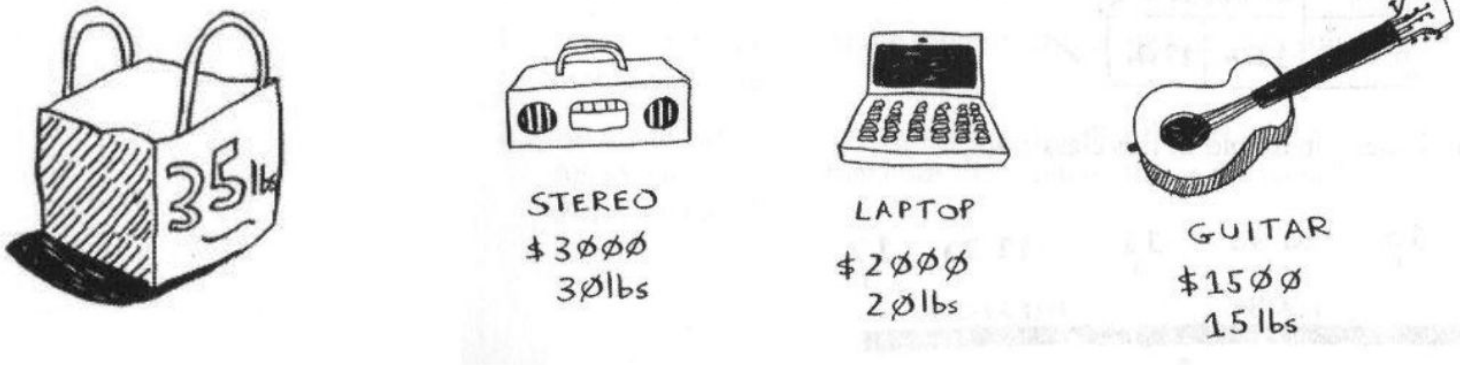
```
def recMinCoin(coinValueList, change, knownResults):
    minCoins = change
    if change in coinValueList:                    #Base Case
        knownResults[change] = 1
        return 1
    elif knownResults[change] > 0:#Case when the result in the table is found, so skip compute step
        return knownResults[change]
    else:                                          #Recursive Case
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMinCoin(coinValueList, change-i, knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins
print(recMinCoin([1, 2, 5, 10], 13, [0]*14))
```

Greedy Choice Property

- Greedy method make whatever choice seems best at the moment and then solve the subproblems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one.

Knapsack Problem

Definition: The knapsack problem is a classic optimisation problem that deals with selecting a subset of items with given weights and values to maximize the total value, subject to a constraint on the total weight.



Types of Knapsack Problems

1. **0/1 Knapsack Problem**
    - Binary decision for each item (either include or exclude).
2. Fractional Knapsack Problem
    - Items can be divided to include fractions.
3. Multiple Knapsack Problem
    - Multiple knapsacks with different capacities.

Greedy Approach

Make locally optimal choices at each stage with the hope of finding a global optimum.

Steps for Greedy Approach:

1. Define a suitable criterion for selection.
2. Make the best choice at each step.
3. Update the problem state.
4. Repeat until a solution is found.

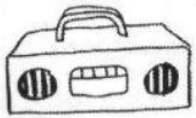
Greedy Approach

1. Define a suitable criterion for selection.
    - Based on value-to-weight ratio, prioritise items with the highest value-to-weight ratio.
2. Make the best choice at each step.
    - Select the highest value-to-weight item first without exceeding the total weight.
3. Update the problem state.
    - Adjust the available capacity and excluding the chosen item from further consideration.
4. Repeat until a solution is found.
    - Until the knapsack is full or no more items can be added due to weight constraints.

Greedy Example 1

Suppose a knapsack has a capacity of 35 lbs and you can only take one of each item (0/1 Knapsack),

- Which of the following items should be included in knapsack to maximise the total value without exceeding the weight capacity?



STEREO
$3000
30lbs

LAPTOP
$2000
20lbs

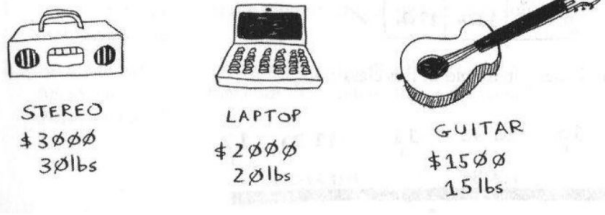
GUITAR
$1500
15 lbs

Greedy Example 1

1. Define criterion for selection.
    - Choose items with the highest value.

2-4. Iterative selection, update and repeat.

1. Choose STEREO (value = 3000$)
    - Update the available capacity to (35 - 30 = 5)
2. Unable to add more items, a solution is found
    - Total value is 3000$
        i. Optimal Solution?



Greedy Example 2

Suppose a knapsack has a capacity of 10 kgs and you can only take one of each item (0/1 Knapsack),

- Which of the following items should be included in knapsack to maximise the total value without exceeding the weight capacity?

Item	Weight (kgs)	Value
A	4	10
B	7	8
C	2	5
D	5	12

Greedy Example 2

1. Define criterion for selection.
    - Choose items with the highest value-to-weight ratio.

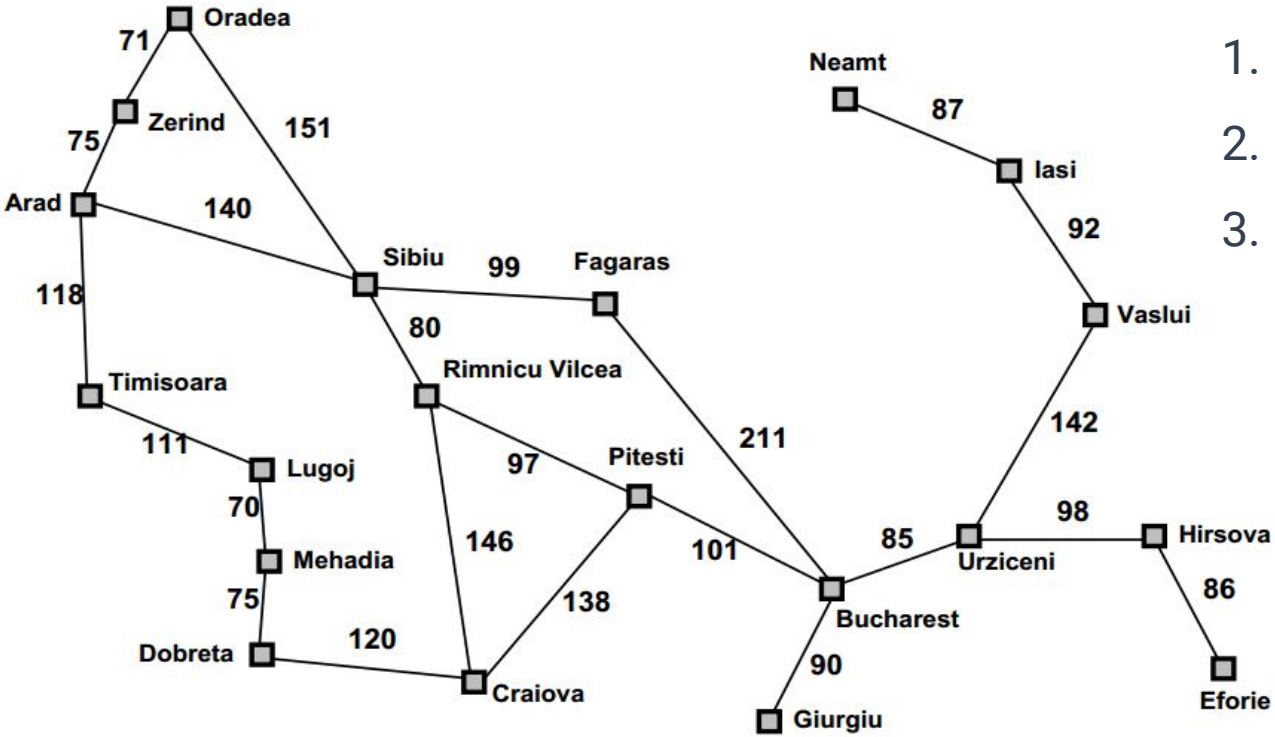
2-4. Iterative selection, update and repeat.

1. Choose item A (value-to-weight ratio = 2.5)
    a. Update the available capacity to (10 - 4 = 6)
2. Choose item C (value-to-weight ratio = 2.5)
    a. Update the available capacity to (6 - 2 = 4)
3. Unable to add more items, a solution is found
    a. Total weight is 4 + 2 = 6
        - Optimal Solution?

Item	Weight (kgs)	Value
A	4	10
B	7	8
C	2	5
D	5	12

Greedy Exercise 1

Find a path from **Arad** to **Bucharest** using Greedy Method



1. What's the criterion?
2. What's the total cost?
3. Does Greedy method provides an optimal path?

Greedy Approach

Advantages of Greedy Approach:

- Simplicity: Greedy algorithms are often simpler to implement.
- Efficiency: Greedy algorithms can be more time-efficient in certain cases.

Limitations of Greedy Approach:

- Optimality: Greedy algorithms may not always guarantee an optimal solution.
- Context Dependency: The choice of criterion is crucial, and it may not work optimally in every scenario.

When Does Greedy Work?

The effectiveness of the Greedy Approach often depends on the nature of the problem. It tends to work well when:

- Optimal Substructure: The problem can be broken down into smaller, **independent** subproblems.
- Greedy Choice Property: A global optimum can be arrived at by selecting a local optimum at each stage.