# Algorithms Analysis and Design

## Chapter 5

## Greedy Technique
## Part 1

# Introduction

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs

- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

**It is well-known that there is no universal technique that can be the best-performing for all problems**

# Intro to optimization problem

# Optimization problem

- An optimization problems can be formulated as follows:

$$
\begin{array}{llll}
\textit{Min / Max} & f(\mathbf{x}) & & \longrightarrow \text{Objective function} \\
\textit{Such that} & g_j(\mathbf{x}) \le 0 & j = 1, \ldots, n & \multirow{2}{*}{$\Big\} \longrightarrow \text{Constraints}$} \\
 & h_k(\mathbf{x}) = 0 & k = 1, \ldots, n &
\end{array}
$$

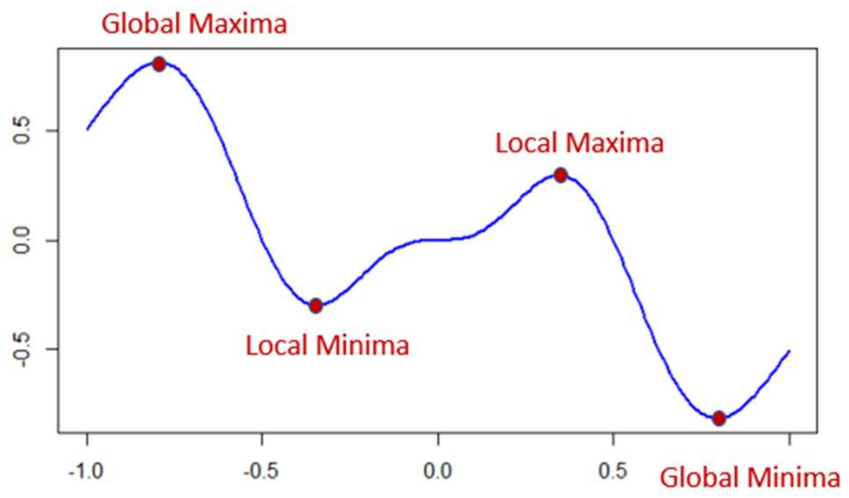
$$\boxed{\begin{array}{c}\text{Find } x = (x_1, x_2, \ldots\ldots\ldots\ x_n) \text{ that maximize / minimize } f(x) \\ \text{considering the given constraints}\end{array}}$$

- Objective function (Fitness): A function used to evaluate every solution of the search space (or assigns score for every solution)
- Feasible solutions: that satisfies all constraints.
- Optimal solution: A feasible solution that maximizing profit / minimizing cost

# Local vs Global optima

- Local optimum: is the best solution to a problem within a small neighborhood of possible solutions

- Global optimum: A solution $s^* \in S$ is a global optimum if it has a better objective function than all solutions of the search space.

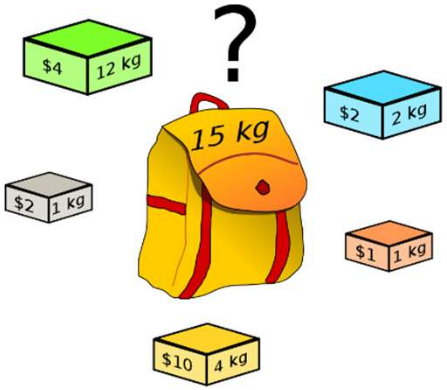



The main goal of solving optimization problems is to find the global optimum ($s^*$)

# Example: 0/1 knapsack problem

- Solution : The set of objects are encoded as a vector of zeros (not selected) and ones (selected)

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 |




- Objective function (profit) : $\sum_{i=1}^{n} X_i \; p_i$
- Feasible solution: $\sum_{i=1}^{n} X_i \; W_i \;\leq\;$ M (capacity)

- Our goal is to find X that maximize $\sum_{i=1}^{n} X_i \; p_i$

subject to : $\sum_{i=1}^{n} X_i \; W_i \;\leq\; M$ and $X_i \in \{ 0 , 1 \}$

# Greedy Technique

# Greedy Technique

Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

- *feasible*
- *locally optimal*
- *irrevocable*

# Greedy Algorithms

- **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.
  - Makes the choice that looks best at the moment in order to get optimal solution.
- Greedy algorithms don't always yield an optimal solution. But sometimes they do.
  - Can be useful for fast approximations
- Used for optimization problems.
- Similar to dynamic programming, but simpler approach (will discuss the differences later)

# General greedy method

```
Greedy ( A , n )
{
        S = ∅
        for i = 1 to n do
            x = select (A)
            if x is feasible
                S = S ∪ x
            endif
        return S
}
```

A : objects
n: number of objects

1. To begin with, the solution set S (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

# Elements of the greedy method

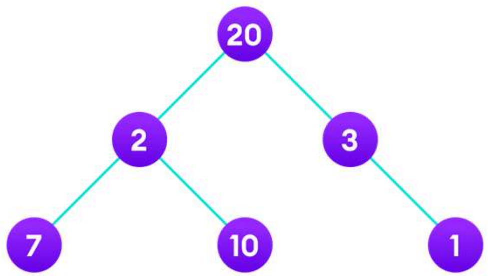
We can determine if the greedy method can be used with any problem if the problem has the following properties. In other words, **to guarantee that a greedy method is correct two things have to be proved:**

1. **Greedy-choice property**: " We can assemble a globally optimal solution by making locally greedy (optimal) choices."
   - Must prove that a greedy choice at each step yields a globally optimal solution.
2. **Optimal substructure**: "an optimal solution to the problem contains within it optimal solutions to subproblems."
   - Global optimal solution is constructed from local optimal solutions.

# Drawback of greedy approach

- As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution.
- **Example**: example, suppose we want to find the longest path in the graph below from root to leaf using greedy approach.

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.
2. the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.
3. Finally the weight of an only child of **3** is **1**. This gives us our final result 20 + 3 + 1 = 24




However, it is not the optimal solution. There is another path that carries more weight 20 + 2 + 10 = 32

# Applications of the Greedy Strategy

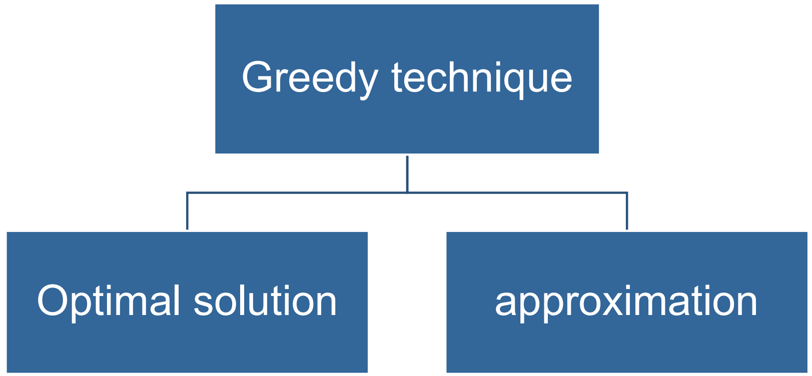
- Greedy technique can be used for handling different kinds of problems:.

- Graph algorithms
  - Minimum spanning trees.
  - Shortest path (Dijkstra's algorithm)
- Scheduling:
  - Activity selection
  - Minimizing time in system
  - Deadline scheduling
- Other:
  - Huffman coding
  - Coloring a graph
  - Traveling Salesman Problem
  - Set-Covering
  - …………

# Applications of the Greedy Strategy

❑ Greedy algorithms don't always yield an optimal solution. But sometimes they do.

- Optimal solutions:
  - change making for "normal" coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes
  - Fractional Knapsack problem

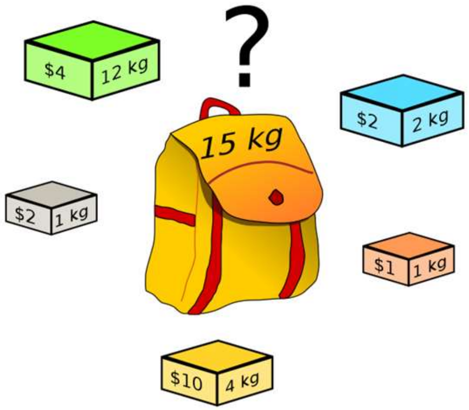



- Approximations:
  - traveling salesman problem (TSP)
  - 0/1 knapsack problem
  - other combinatorial optimization problems

# Applications of the Greedy Strategy

In this course we will discuss some selected problems that can be solved by greedy technique

- Fractional Knapsack problem.
- Minimum Spanning Tree (Kruskal and prim's algorithms).
- Shortest Path (Dijkstra's algorithm).
- Coin Changing problem
- Lossless data compression (Huffman code).
- Simple scheduling problems.

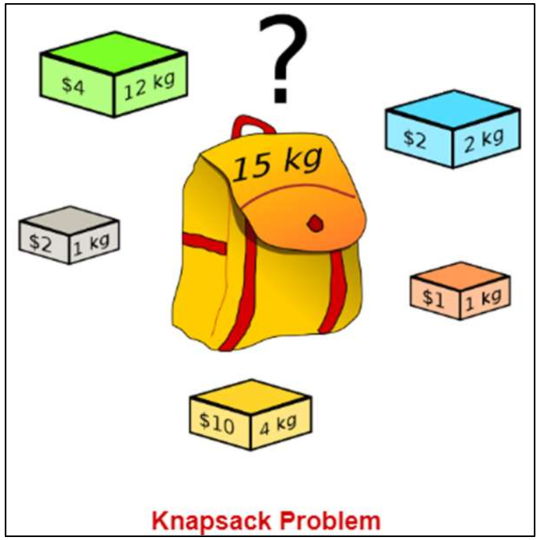



# Fractional Knapsack Problem

# Knapsack Problem

- **Knapsack problem**: Given $n$ items $I_1, I_2, \ldots\ldots, In$ of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity $W$

  find **the most valuable subset of the items** that fit into the knapsack.




- Which items should be placed into the knapsack such that:
  - The value or profit obtained by putting the items into the knapsack is maximum.
  - And the weight limit of the knapsack **does not exceed**.

- 0/1 knapsack problem
- Fractional Knapsack problem

# Fractional vs 0/1 Knapsack Problem

- **Recall**: in Chapter 3 we discussed the 0/1 knapsack problem
- **0/1 Knapsack Problem**:

  This problem is solved efficiently by using a ***dynamic programming***

  - There is no possible to take a fractional amount of an item.
  - Either take it completely or leave it completely.
  - a binary (0-1) choice for each item: '**0**' means that we are not taking that item and '**1**' means that we are taking the item.
- **fractional Knapsack Problem**:

  This problem is solved by using a ***greedy approach***

  - can take any fraction of an item

**0/1 Knapsack Problem**

find X such that for all $x_i = 0 , 1$, $i = 1, 2, .., n$

$\sum w_i x_i \le W$ and

$\sum x_i v_i$ is maximum

**Fractional Knapsack Problem**

find X such 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

- Knapsack capacity: $W$
- There are $n$ items: the $i$-th item has value (or profit) $v_i$ and weight $w_i$

| item | 1 | 2 | …… | i | …… | n |
|---|---|---|---|---|---|---|
| weights | $w_1$ | $w_2$ | …… | $w_i$ | …… | $w_n$ |
| value | $v_1$ | $v_2$ | …… | $v_i$ | …… | $v_n$ |

- **Goal**:
  - find X such that for all $0 \leq x_i \leq 1$, $i = 1, 2, .., n$ ⟶ A vector $(x_1, x_2, ........x_i)$

constraint

$\sum w_i x_i \leq W$ ⟶ $w_1 * x1 + w_2 * x2 + \ldots\ldots. w_n * xn \leq W$

function

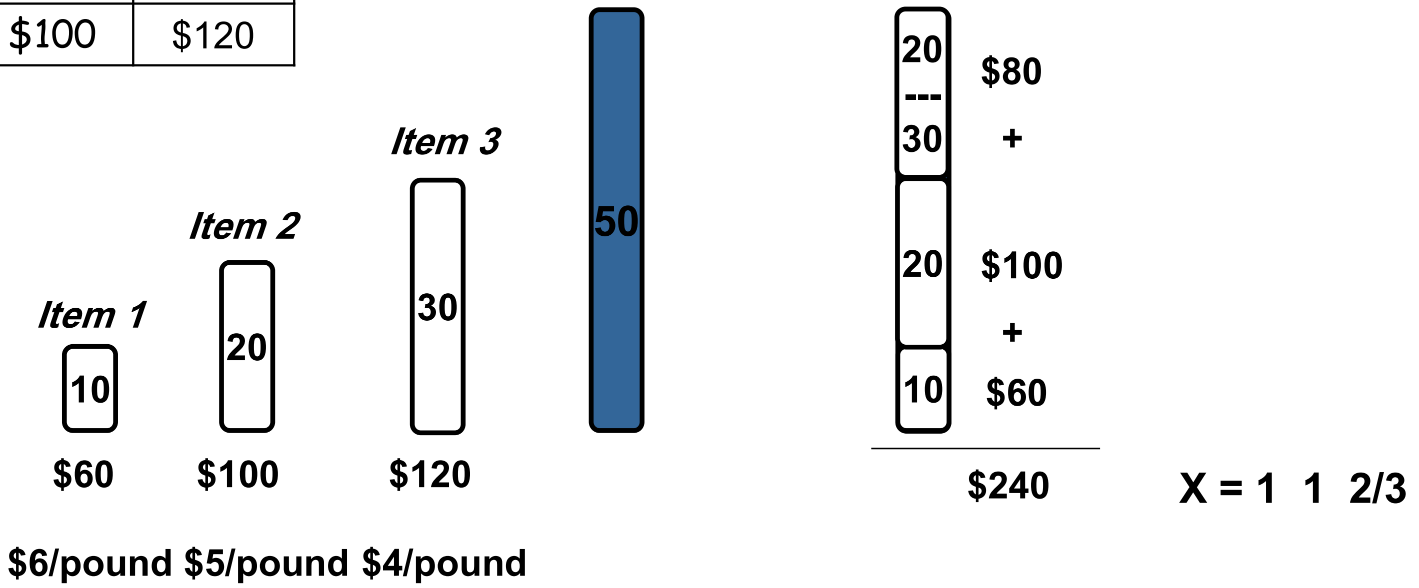
$\sum x_i v_i$ is maximum

# Fractional Knapsack - Example

- *E.g.:*

| item | 1 | 2 | 3 |
|---|---|---|---|
| weights | 10 | 20 | 30 |
| value | $60 | $100 | $120 |

Knapsack capacity: W = 50



# Greedy method for Fractional Knapsack Problem

- Greedy strategy 1:
  - **Pick the item with the maximum value**

This approach does not produce the optimal solution

- *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) = $v_2/w_2 = 1$

| item | 1 | 2 |
|---|---|---|
| weights | 100 | 1 |
| value | 2 | 1 |

# Greedy method for Fractional Knapsack Problem [Cont]

**Greedy strategy 2:**

This approach produces the optimal solution

- **Pick the item with the maximum value per weight $v_i/w_i$ (we can call this value density)**
- 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

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

# Example

Knapsack capacity: W = 50

| item | 1 | 2 | 3 |
|---|---|---|---|
| w | 10 | 30 | 20 |
| v | 60 | 120 | 100 |
| v/w | 6 | 4 | 5 |

**Steps**:

1. Calculate density $\mathbf{d_i = v_i/w_i}$ (value/profit per unit) for each item
2. Sort the items based on their densities (descending order)
3. Select items until capacity is full

| item | 1 | 3 | 2 |
|---|---|---|---|
| v/w | 6 | 5 | 4 |

X = 1 $\frac{2}{3}$ 1 The optimal solution

Total value = 1 * 60 + 2/3 * 120 + 1 * 100
= 60 + 80 + 100 = 240

# Pseudocode of Greedy algorithm for Fractional Knapsack Problem

ALGORITHM $Fractional - Knapsack$ (W, v[n], w[n])

1. calculate density value $d_i$ for each item
2. sort items in descending order based on density values
3. While w > 0 and as long as there are items remaining
4. pick item with maximum $v_i/w_i$
5. $x_i \leftarrow \min(1, w/w_i)$
6. remove item $i$ from list
7. $w \leftarrow w - x_i w_i$

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

# Analysis of Greedy algorithm for Fractional Knapsack Problem

1. Calculate density $\mathbf{d_i = v_i/w_i}$ (value/profit per unit) for each item — $\Theta(n)$
2. Sort the items based on their densities (descending order) — $\Theta(n \log n)$
3. Select items until capacity is full — $\Theta(n)$

Running tim $T(n) = \Theta(n) + \underbrace{\boldsymbol{\Theta(n\lg n)}}_{\text{Leading term}} + \Theta(n)$

Time complexity is $\Theta(n \log n)$

The main time taking step is the **<u>sorting</u>** of all items in decreasing order of their value / weight ratio

# Analysis of Greedy algorithm for Fractional Knapsack Problem

ALGORITHM $Fractional - Knapsack$ (W, v[n], w[n])

1. calculate density value $d_i$ for each item → $\Theta(n)$
2. sort items in descending order based on density values → $\Theta(n \log n)$
3. While w > 0 and as long as there are items remaining
4. pick item with maximum $v_i/w_i$
5. $x_i \leftarrow \min(1, w/w_i)$
6. remove item $i$ from list
7. $w \leftarrow w - x_i w_i$

$\Theta(n)$

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

# Example

Assume that we have a knapsack with max weight capacity, $\boldsymbol{W} = \mathbf{16}$. Assuming you can take fractions of items, apply greedy method to fill the knapsack with items such that the benefit (value or profit) is maximum without crossing the weight limit W.

| item | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| w | 6 | 10 | 3 | 5 | 1 | 3 |
| v | 6 | 2 | 1 | 8 | 3 | 5 |
| d = v/w | 1 | 0.2 | 0.333 | 1.6 | 3 | 1.667 |

1. Compute density for each item.
2. Sort items in descending order as per density value
3. Fill the table below

Initially w = 16

| | $item$ | $weight$ | $value$ | $density\ v/w$ | $Xi = \min(1, w/wi)$ | $w = w - (xi * wi)$ | $Profit\ (xi * vi)$ |
|---|---|---|---|---|---|---|---|
| | 5 | 1 | 3 | 3 | Min (1, 16) = 1 | 15 | 3 |
| | 6 | 3 | 5 | 1.667 | Min (1, 5) = 1 | 12 | 5 |
| | 4 | 5 | 8 | 1.6 | Min (1, 12/5) = 1 | 7 | 8 |
| | 1 | 6 | 6 | 1 | Min (1, 7/6) = 1 | 1 | 6 |
| | 3 | 3 | 1 | 0.333 | Min (1, 1/3) = 1/3 | 0 | 1/3 |
| ignored | 2 | 10 | 2 | 0.2 | 0 | | |

X = 1 0 1/3 1 1 1 Total profit = **22.33**

# Exercise

Assume that we have a knapsack with max weight capacity, $\boldsymbol{W} = \mathbf{16}$. Apply the greedy method to fill the knapsack with items such that the benefit (value or profit) is maximum without crossing the weight limit W:

a) Assuming you can take fractions of items.

b) Assuming you can't take fractions of items (0/1 knapsack problem)

Compared to the optimal solution obtained by brute force method (**see the same example in chapter 3**). Does the greedy method provides the optimal solution for 0/1 knapsack problem?

Knapsack capacity W=16

| item | weight | value |
|---|---|---|
| 1 | 2 | $20 |
| 2 | 5 | $30 |
| 3 | 10 | $50 |
| 4 | 5 | $10 |

# Exercise

Assume that we have a knapsack with max weight capacity, $\boldsymbol{W = 16}$. Apply the greedy method to fill the knapsack with items such that the benefit (value or profit) is maximum without crossing the weight limit W:

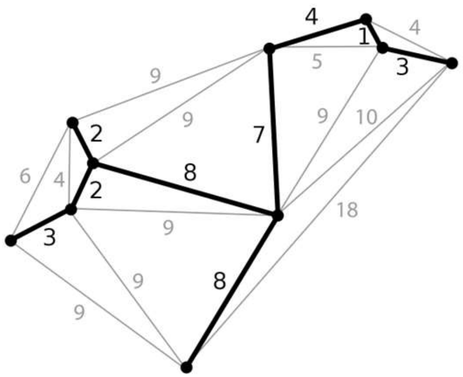
a) Assuming you can take fractions of items.

b) Assuming you can't take fractions of items (0/1 knapsack problem)

Compared to the optimal solution obtained by brute force method (**see the same example in chapter 3**). Does the greedy method provides the optimal solution for 0/1 knapsack problem?

Knapsack capacity W=16

| item | weight | value |
|---|---|---|
| 1 | 2 | $20 |
| 2 | 5 | $30 |
| 3 | 10 | $50 |
| 4 | 5 | $10 |




# Coin Change problem

# Coin Change Problem

- **Given** unlimited amounts of coins of denominations $d1 > \ldots > dm$
- ***Goal***: make change for amount $V$ with the least number of coins.

have to find the minimum number of coins which satisfies the value $V$

**Example**: suppose you have the following denominations of coins ***{5,10,20,25}*** *and a certain amount of change* ***V = 50. How to use the fewest coins to make this change?***

***V = 25 a + 20 b + 10 c + 5 d***

***what are a , b , c , d minimizing (a+b+c+d)?***

**Possible Solutions**

{coin * count}
{5 * 10} = 50 [10 coins]
{5 * 8 + 10 * 1} = 50 [9 coins].
{10 * 5} = 50 [5 coins]
{20 * 2 + 10 * 1} = 50 [3 coins]
{20 * 2 + 5 * 2} = 50 [4 coins]
**{25 * 2} = 50 [2 coins]** **best solution (number of coins = 2)**

# Greedy method for coin change problem

$O(n \log n)$

**Greedy strategy:**

**Pick the denomination with the maximum value …..**

- Sort the denominations in descending order based on their values. $\rightarrow n \log n$
- Pick the largest denomination that is smaller than the current amount. $\rightarrow n$
- Add selected denomination to result and subtract its value from amount.
- Repeat until the remaining amount becomes 0.

Does this approach yield to the optimal solution??

# Greedy method for coin change problem

**Optimal substructure:**

- After the greedy choice, assuming the greedy choice (i.e. chose the most valuable coin) is correct, **can we get the optimal solution from sub optimal result?**
- Example: collection of change 25 , 10 , 5 , 1    V = 38

Assuming we choose 25 then the optimal solution of 38 = 25 + optimal coin (38-25)

**Greedy Choice Property:**

- If we do not choose the most valuable coin, is there a better solution?

# Greedy method for coin change problem

**Greedy Choice Property:**

- If we do not choose the most valuable coin, is there a better solution?

  - You are given the collection of change (**25**) (**10**) (**5**) (**1**), and a value **A**=92

    `25*3 + 10*1 + 5*1 + 1*2`

    Using only **7** coins

    The greedy choice is not violated

  - If you are given the collection of change **(12)(5)(1)** and a value **A**=15

    `12*1+1*3 using 4 coins`

    **`but there is a better solution`**

    `15=5*3 using only 3 coins`

    The greedy choice is violated

**The greedy algorithm doesn't always give the best solution, So <u>the greedy choice property is not correct</u>.**

# Pseudocode of Greedy algorithm for Coin Change Problem

```
ALGORITHM Coin_Change (D[n] , V)
        Sort coin denominations  in descending order
        S = ∅
        For i = 1 to n do
            while V >= D[i] do
                S = S ∪ D[i]
                V = V - D[i]
            if V = 0 break
        Return S
```

# Pseudocode version2

ALGORITHM *Coin_Change* (D[n] , V)

Sort coin denominations in descending order $\Big]\, n\log n$

$i = 1$

**While ( V > 0 )**

X[i] = V / D[i]

V = V - X[i] * D[i] $\quad n$

i = i + 1

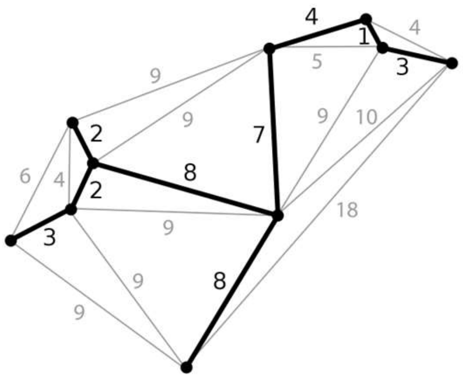
**Return** X[1] + X[2] + …….. X[n] / number of coins

Time complexity: $\Theta$(nlgn)

What is the time complexity if it is assumed that the input coins are already sorted??????

# Notes

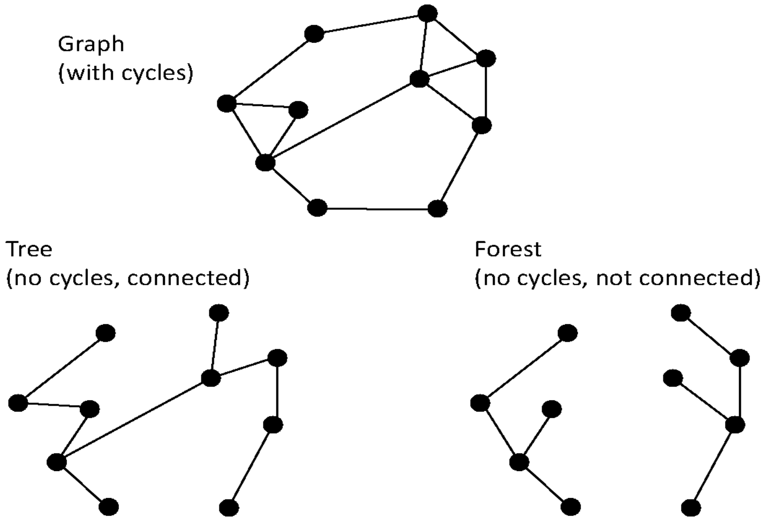
- We can solve this problem by exhaustively enumerating the feasible solutions (**Brute Force**) and selecting the one with the fewest number of coins→ this is an **exponential time** algorithm
- The optimal solution to the coin change problem can be computed in **feasible time** using dynamic programming.




# Minimum spanning tree

# Trees

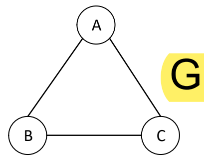
- A **tree** is a **connected acyclic** graph.
  - **connected**: if for every pair of its vertices u and v there is a path from u to v.
  - **Acyclic**: A graph with **no cycles**.
- A **forest**, A graph that has **no cycles** but is not necessarily connected (**disconnected**).



# Spanning Tree

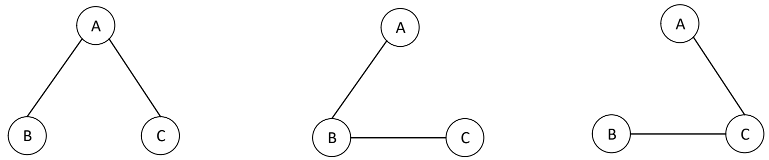
O(nlogn)

- A ***spanning tree*** (*T*) of an undirected connected graph ***G*** is its connected acyclic subgraph of **G** that includes all G's vertices.
  - No loops.
  - Connected.
  - $| E | = | V | - 1$ (number of edges = number of vertices – 1).
  - Spanning tree must contain the same number of vertices as of graph G
- In general, a graph may have several spanning trees




Graph (G)

3 Spanning tree structures



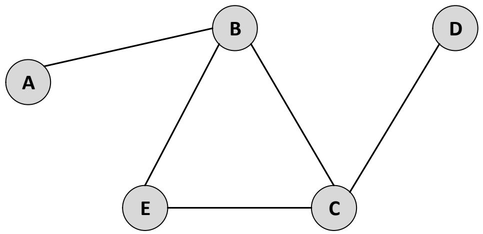
# Example

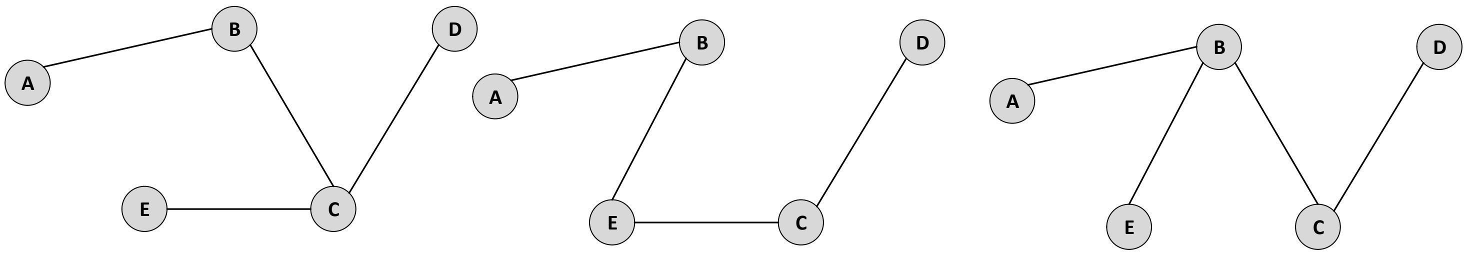
❑ **Example**: Let **G** be an undirected graph, find and draw all spanning trees of G

Graph (G) :

Vertices = 5

Edges = 5




Spanning Trees

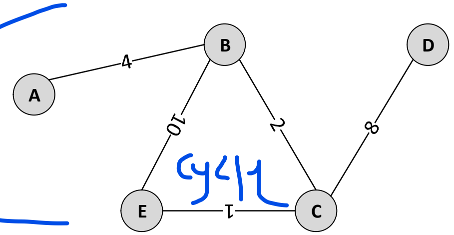
Vertices = 5

Edges = 4

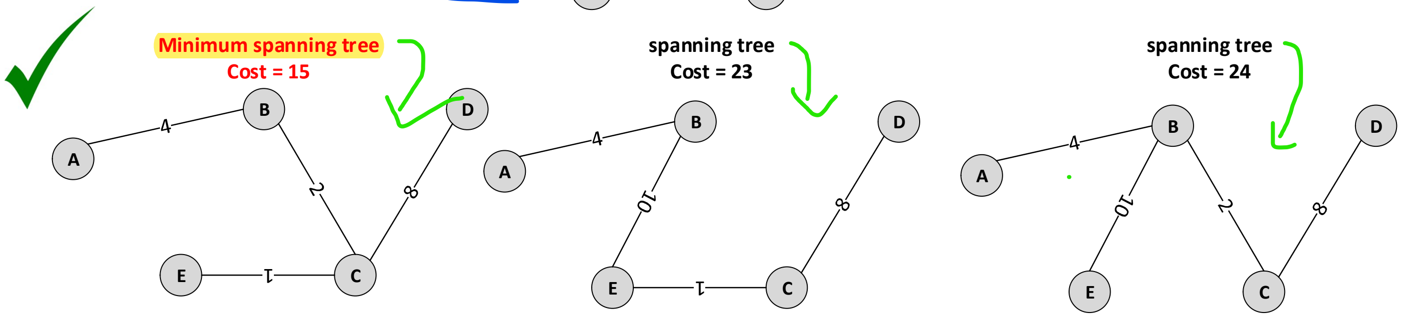
# Minimum Spanning Tree (MST)

- For an undirected connected graph ***G*** that has weighted assigned to its edges, a ***minimum spanning tree*** is its **spanning tree of the smallest weight**, where the weight of a tree is defined as the sum of the weights on all its edges.

tree




$|E| = |V| - 1$




The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph

# Applications of MST

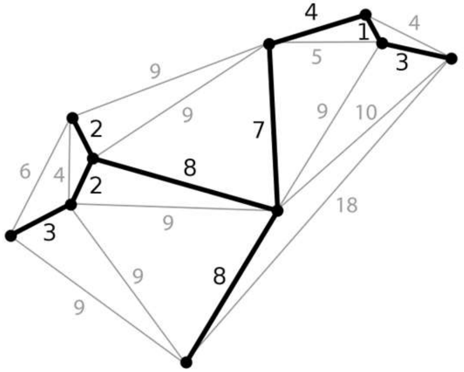
MST problem arises naturally in many practical situations: **given $n$ points , connect them in the cheapest possible way so that there will be a path between every pair of points**. We can represent the points given by vertices of a graph, possible connections by the graph's edges, and the connection costs by the edge weights.

- It has direct applications to *the design of all kinds of networks*— including **telecommunication** networks, **computer** networks, **transportation** networks, and **electrical** grids, and **water supply** networks—by providing the cheapest way to achieve connectivity.
- It is also helpful for constructing approximate solutions to more difficult problems such the **traveling salesman problem**.
- It identifies clusters of points in data sets. It has been used for classification purposes in archeology, biology, sociology, and other sciences.

# Greedy methods for finding MST

- There are Several efficient algorithms available for handling MST problem that **yield the optimal solution**. In this section, we will outline two well-known algorithms:
  - **Kruskal's algorithm**
  - **Prim's algorithm.**

These two algorithms yield an optimal solution




# Kruskal’s Algorithm

# Kruskal's algorithm

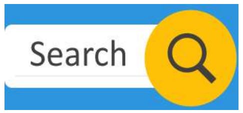
- Begin by sorting the graph's edges in nondecreasing order of their weights.
- Starting with the empty subgraph, it scans the sorted list and apply the greedy rule
  - Add an edge of min weight to the current subgraph that <u>does not make a cycle</u>.
  - Skip the edge otherwise.

Greedy-choice property

- Continue until you get a single tree **T**
  - **T** contains **|V|-1** edges

# To implement Kruskal's algorithm

- We must select the edges in increasing order of weight
  - Sort or Min-Heap log n
- We must be able to determine whether adding an edge will create a cycle
  - There is an efficient algorithm for doing so called Union-Find




**Note**: It is possible to use the **Min-Heap** to select the edge with minimum weight at each step

$E \log E$

# Pseudocode of Kruskal's algorithm

**ALGORITHM** *Kruskal(G)* → graph

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
sort | sort $E$ in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$ → $E \log E$
$E_T \leftarrow \varnothing$; $\quad ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
**while** $ecounter < |V| - 1$ **do**
  $k \leftarrow k + 1$
  **if** $E_T \cup \{e_{i_k}\}$ is acyclic   union_find → $O(EV)$
    $E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
**return** $E_T$

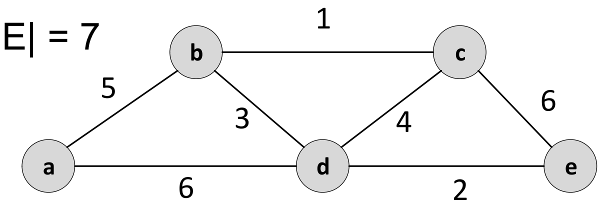
# Time Complexity of Kruskal’s algorithms

- **With an efficient union-find algorithm**, the running time of Kruskal’s algorithm will be **dominated by the time needed for sorting** the edge weights of a given graph.
- Hence, with an efficient sorting algorithm, the time efficiency of Kruskal’s algorithm will be in $O(|E| \log |E|)$.

# Application of Kruskal's algorithms

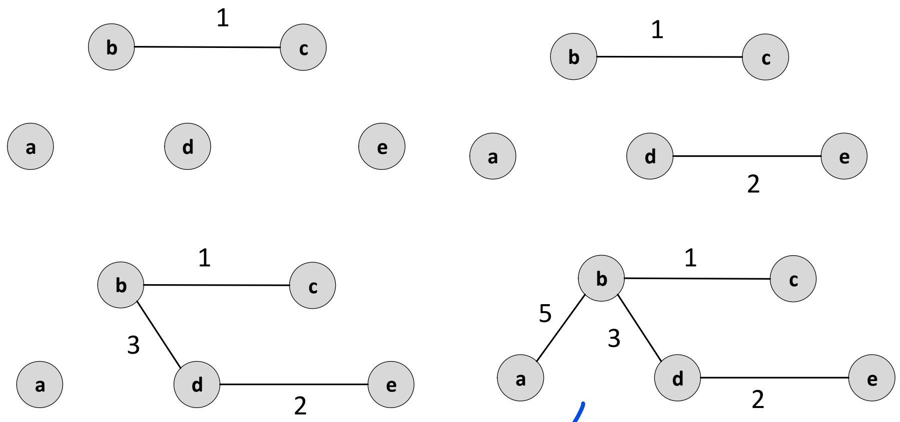
**Example**: Apply Kruskal's algorithm to find a minimum spanning tree of the following graph.

$|V| = 5$ $|E| = 7$




| | 1 | 2 | 3 | 4 | 5 | 6 | 6 |
|---|---|---|---|---|---|---|---|
| Sorted list of edges: | bc | de | bd | cd | ab | ad | ce |

| edges | weight | action |
|---|---|---|
| bc | 1 | accept |
| de | 2 | accept |
| bd | 3 | accept |
| cd | 4 | reject |
| ab | 5 | accept |
| Stop : $\lvert E\rvert = 4$ | Cost = 11 | |



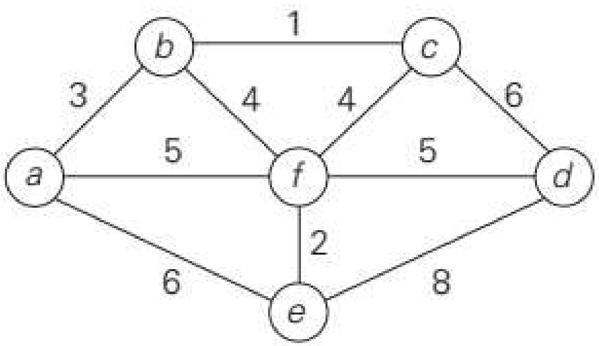

MST

$|E| = |V| - 1$ ✓

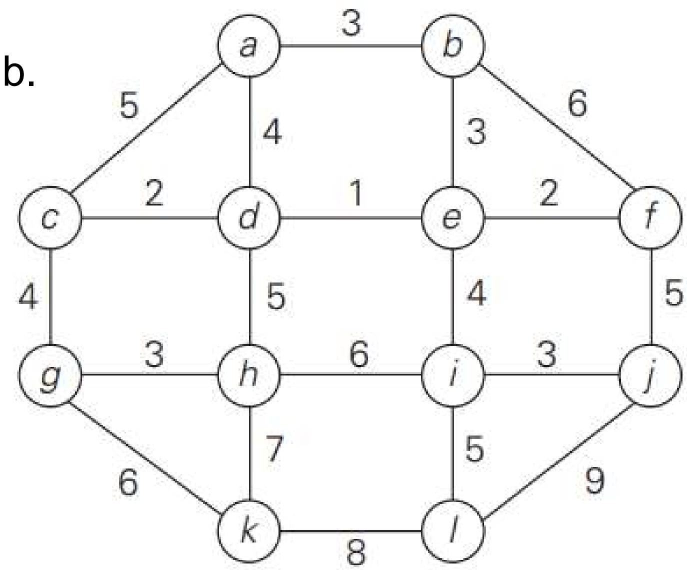
# Application of Kruskal's algorithms

**Exercise**: Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs (**show your work**).

a.




b.



# Exercises

1) Does Kruskal's algorithm work correctly on graphs that have negative edge weights? Explain why?

2) Design an algorithm for finding a ***maximum spanning tree***—a spanning tree with the largest possible edge weight—of a weighted connected graph.

# Animations of MST algorithms

For a better understanding of Kruskal’s, Prim’s algorithms and visualizing its operation through animation:

https://visualgo.net/en/mst?slide=1

https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

https://www.cs.usfca.edu/~galles/visualization/Prim.html