



# Greedy and Dynamic Programming

**Dr. Gopal Chandra Jana**  
**Assistant Professor**

# INTRODUCTION

- Greedy and Dynamic Programming are two powerful techniques in algorithm design.
- They are used to solve **optimization problems efficiently**.
- Understanding their principles and differences is crucial for tackling a wide range of computational problems.
- Also to consider certain parameters before applying any of these.

# Choice Between Greedy and Dynamic Programming

- Greedy algorithms are preferred when the problem exhibits the **greedy-choice property** and **optimal substructure**.
- Dynamic Programming is suitable when the problem can be broken down into **overlapping sub-problems** with **optimal substructure**.
- Understanding the **problem's characteristics is essential** for selecting the appropriate technique.



Deciding whether to apply a greedy or dynamic programming approach to solve a problem depends on **several key parameters and characteristics of the problem**. Here are some factors to consider:

# 1. Optimal Substructure:

Greedy:

- Optimal solution can be constructed from locally optimal choices.
  - **Eg:** Finding the shortest path in a graph using **Dijkstra's algorithm**.

Dynamic Programming:

- Optimal solution can be derived from optimal solutions to sub-problems.
  - **Eg:** Calculating the nth Fibonacci number using **memoization**.

## 2. Overlapping Sub-problems:

Greedy:

- If the problem **does not have overlapping sub-problems**, a greedy approach is usually sufficient.
  - Eg: **Coin Change Problem** where each coin can only be used once.

Dynamic Programming:

- If the problem **can be broken down into overlapping sub-problems**, dynamic programming is more appropriate.
  - Eg: **Coin Change Problem** where coins can be reused to make change.

# 3. Greedy Choice Property:

Greedy:

- Makes locally optimal choices expecting a globally optimal solution.
  - **Eg: Fractional Knapsack Problem, where items can be divided and selected based on their value-to-weight ratio.**

Dynamic Programming:

- Evaluates all possible choices for optimal solution.
  - **Eg: 0/1 Knapsack Problem, where items cannot be divided, and a subset must be chosen to maximize value without exceeding the weight limit.**

# 4. Complexity and Efficiency:

Greedy:

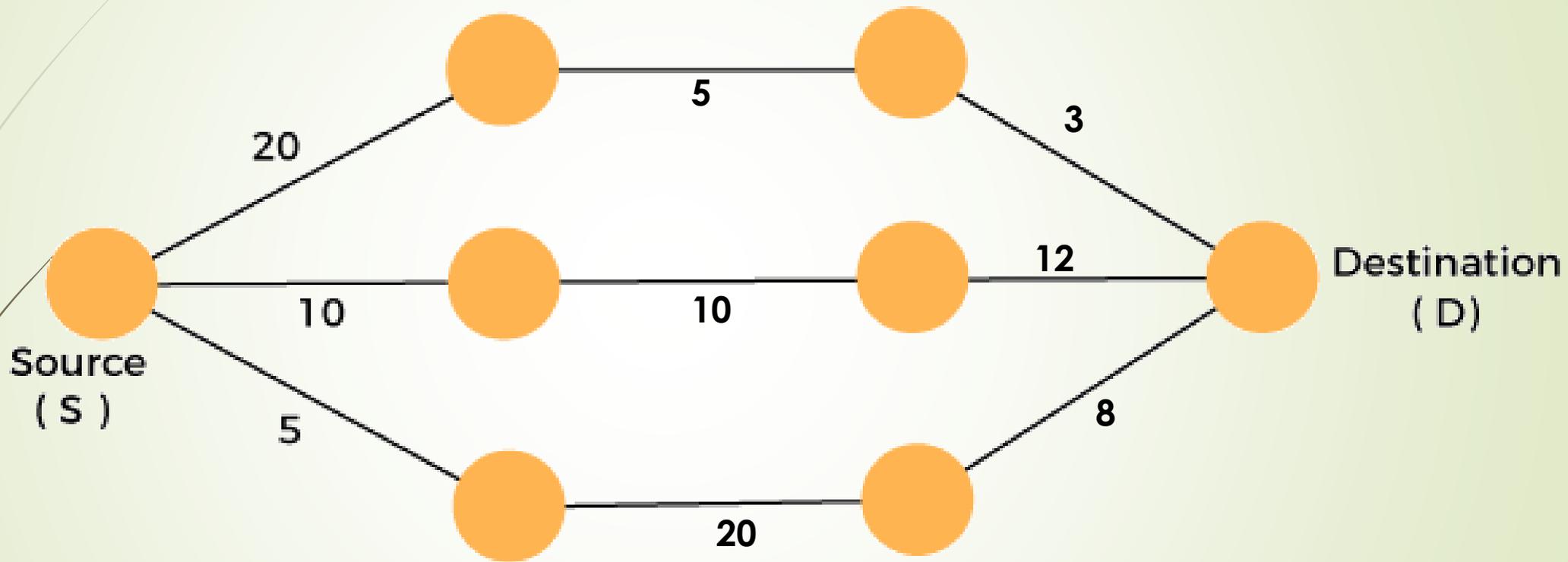
- Simple and efficient but **may not always provide optimal solution**.
  - **Eg: Activity Selection Problem**, where activities with the earliest finish times are chosen iteratively.
- Typically **requires less memory**.
  - **Eg: Huffman Coding for data compression**, which constructs a binary tree based on the frequency of characters.

Dynamic Programming:

- Dynamic programming can **guarantee the optimal solution** but may be more **computationally expensive**.
  - **Eg: Longest Common Subsequence Problem**, where all possible subsequences must be examined to find the longest common subsequence.
- May require **more memory due to storing solutions to sub-problems**.
  - **Eg: Matrix Chain Multiplication Problem**, which involves finding the most efficient way to multiply matrices.

# Greedy Algorithm

- Greedy algorithms make decisions based on the **current best choice** without considering future consequences.
- They aim to find the globally optimal solution by making a series of locally optimal choices.
- Typically efficient and easy to implement.
- **Examples:** Dijkstra's algorithm for shortest paths  
Kruskal's algorithm for minimum spanning trees.



# Greedy Problems:

- **Fractional Knapsack Problem:** Given items with weight and value, determine the maximum value of fractions of the items that can be taken into a knapsack of limited weight capacity.
- **Activity Selection Problem:** Given a set of activities with start and finish times, select the maximum number of non-overlapping activities that can be performed by a single person.
- **Huffman Coding:** Given a set of characters and their frequencies, construct a binary tree such that the encoded binary codes for characters have minimum total length.
- **Coin Change Problem:** Given a set of coin denominations and a target amount, find the minimum number of coins needed to make up the target amount.
- **Job Sequencing Problem:** Given a set of jobs with deadlines and profits, find the maximum profit subset of jobs that can be completed within their deadlines.
- **Minimum Spanning Tree Algorithms:** Algorithms like Prim's and Kruskal's are greedy approaches to find the minimum spanning tree in a connected, undirected graph.
- **Dijkstra's Algorithm:** To find the shortest path between nodes in a graph with non-negative edge weights.

# Fractional Knapsack

Given the weights and values of  $N$  items, in the form of {weight, Value} put these items in a knapsack of capacity  $W$  to get the maximum total profit in the knapsack. In Fractional Knapsack, we can break items for maximizing the total value of the knapsack.

Input: arr[] = {{5, 30}, {10, 20}, {20, 100}, {30, 90}, {40, 160}},  $W = 60$

Output: 270

item	$w_i$	$v_i$	weight Ratio $P_i$
I1	5	30	6
I2	10	20	2
I3	20	100	5
I4	30	90	3
I5	40	160	4

Capacity of Knapsack is  
 $= 60$

item	$w_i$	$v_i$	$P_i$
I1	5	30	6
I3	20	100	5
I5	40	160	4
I4	30	90	3
I2	10	20	2

$$\text{Net Selected Value} = I1 + I3 + I5 \cdot \frac{35}{40} = 30 + 100 + 160 \cdot \frac{35}{40} = 270$$

# Activity Selection Problem

eg:

$i$	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

Greedy Activity selector ( $s, f$ )

$n \leftarrow \text{length}(s)$

$A \leftarrow \{a_1\}$

$i \leftarrow 1$

for  $m \leftarrow 2$  to  $n$

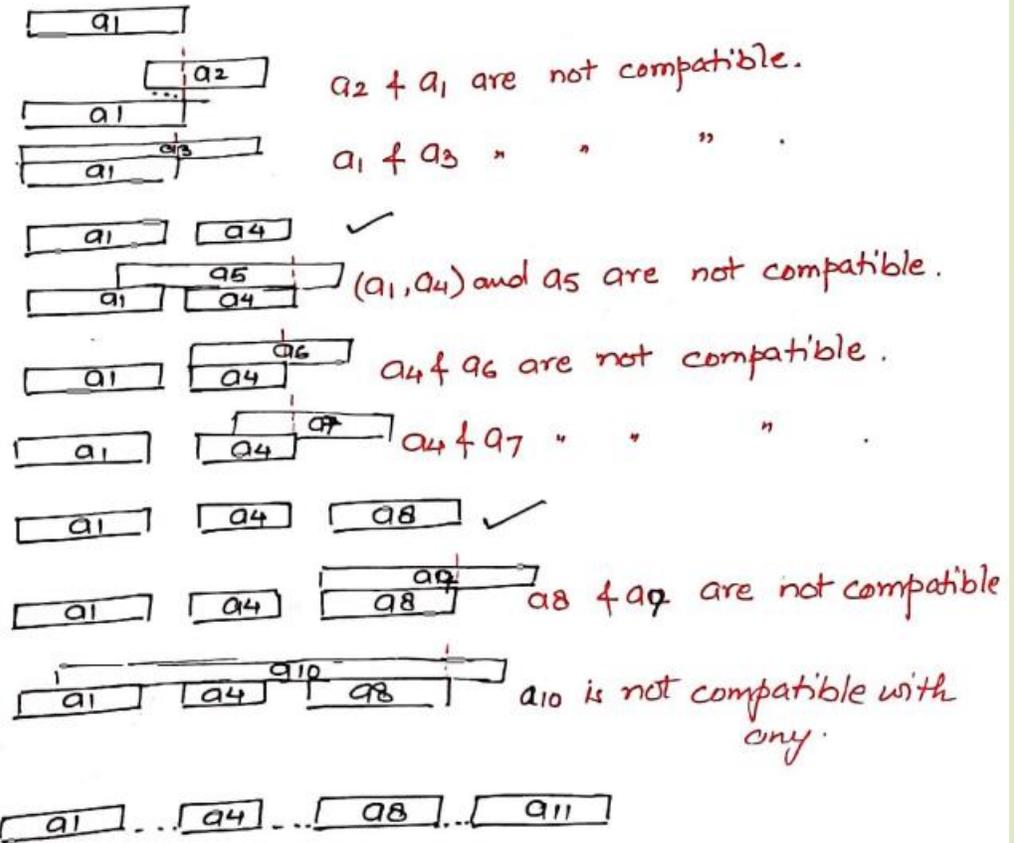
do if  $s_m \geq f_i$

then  $A \leftarrow A \cup \{a_m\}$

$i \leftarrow m$

return  $A$ .

$i$	$s_i$	$f_i$
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



The resulting set of selected activities is  $\langle a_1, a_4, a_8, a_{11} \rangle$

# Huffman DeCode

```
//Function to return the decoded string.  
string decodeHuffmanData(struct MinHeapNode* root, string s)  
{  
    // Code here  
    string ans = "";  
    struct MinHeapNode* curr = root;  
    for (int i = 0; i < s.size(); i++) {  
        if (s[i] == '0')  
            curr = curr->left;  
        else  
            curr = curr->right;  
  
        // reached leaf node  
        if (curr->left == NULL and curr->right == NULL) {  
            ans += curr->data;  
            curr = root;  
        }  
    }  
    // cout<<ans<<endl;  
    return ans + '\0';  
}
```

# Coin Change -Greedy

Given a value of V Rs and an infinite supply of each of the denominations {1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, The task is to find the minimum number of coins and/or notes needed to make the change?

## Examples:

Input: V = 121

Output: 3

**Explanation:** We need a 100 Rs note, a 20 Rs note, and a 1 Rs coin.

1. Declare a vector that store the coins.
2. while n is greater than 0 iterate through greater to smaller coins:
  1. if n is greater than equal to 2000 than push 2000 into the vector and decrement its value from n.
  2. else if n is greater than equal to 500 than push 500 into the vector and decrement its value from n.
  3. And so on till the last coin using ladder if else.

# Coin Change -Greedy

Given a value of  $V$  Rs and an infinite supply of each of the denominations  $\{1, 2, 5, 10, 20, 50, 100, 500, 1000\}$  valued coins/notes, The task is to find the minimum number of coins and/or notes needed to make the change?

## Examples:

Input:  $V = 121$

Output: 3

**Explanation:** We need a 100 Rs note, a 20 Rs note, and a 1 Rs coin.

- Sort the array of coins in decreasing order.
- Initialize **ans** vector as empty.
- Find the largest denomination that is smaller than **remaining amount** and while it is smaller than the **remaining amount**:
  - Add found denomination to **ans**. Subtract value of found denomination from **amount**.
- If amount becomes **0**, then print **ans**.

# Coin Change -Greedy

```
// C++ program to find minimum
// number of denominations
#include <bits/stdc++.h>
using namespace std;

// All denominations of Indian Currency
int denomination[]
    = { 1, 2, 5, 10, 20, 50, 100, 500, 1000 };
int n = sizeof(denomination) /
        sizeof(denomination[0]);
```

```
void findMin(int V)
{
    sort(denomination, denomination + n);
    // Initialize result
    vector<int> ans;
    // Traverse through all denomination
    for (int i = n - 1; i >= 0; i--) {
        // Find denominations
        while (V >= denomination[i]) {
            V -= denomination[i];
            ans.push_back(denomination[i]);
        }
    }
    for (int i = 0; i < ans.size(); i++)
        cout << ans[i] << " ";
}
```