MODULE IV

Computational Approaches to Problem-Solving

Computational approaches to problem-solving use the power of computers to solve complex problems efficiently. This module covers key strategies such as:

- **Brute Force**: Trying all possible solutions to find the correct one.
- **Divide-and-Conquer**: Breaking a problem into smaller parts, solving each, and combining the results.
- **Dynamic Programming**: Solving problems by reusing solutions to smaller subproblems.
- **Greedy Algorithms**: Making the best choice at each step to find a solution.
- **Randomized Approaches**: Using randomness to explore solutions in an unpredictable way.

BRUTE-FORCE APPROACH TO PROBLEM SOLVING

The Brute-Force Approach is a basic, straightforward method for solving problems by systematically testing all possible solutions until the correct one is found. While it guarantees finding the solution if one exists, it is often inefficient due to its exhaustive nature.

It relies on trying every possible combination to find the correct result.

This approach is commonly used in scenarios where:

- The solution space is small.
- Simplicity is preferred over efficiency.
- No better algorithm is readily available.

Characteristics of Brute-Force Solutions

1. Exhaustive Search: Every possible solution is examined without any optimization.
2. Simplicity: Easy to understand and implement.
3. Inefficiency: Often slow due to the large number of possibilities.
4. Guaranteed Solution: If a solution exists, the brute-force method will eventually find it.

Examples:

1. Padlock Combination:

- Imagine a padlock with 3 dials, each having 10 digits (0-9).
- The brute-force approach would involve trying all 1,000 combinations (000 to 999) until the correct one unlocks the padlock.
- Steps:
    1. Start with 000.
    2. Increment sequentially: 001, 002, ..., 999.
    3. Stop when the lock opens.

Program:

```
# Brute-force approach to unlock a padlock
def padlock(key):
    for combination in range(1000):  # 000 to 999
        guess = f"{combination:03}"  # Format to always have 3 digits
        print(f"Trying combination: {guess}")
        if guess == key:
            print(f"Padlock opened with combination: {guess}")
            break

# Example usage
padlock(“302”)
```

2. Password Guessing

In cybersecurity, brute-force attacks involve systematically trying every possible password combination until the correct one is found. This method works well on weak passwords, such as short or simple ones. For example, cracking a 6-character password made up of letters, digits, and special characters requires testing all **217.8 trillion** possible combinations (95^6) until the correct one is discovered.

Here, **95** is the total number of possible characters (26 lowercase + 26 uppercase + 10 digits + 33 special characters), and 95^6 gives the total number of possible combinations for a 6-character password.

Python Program: (For a 3-letter alpha-numeric password)

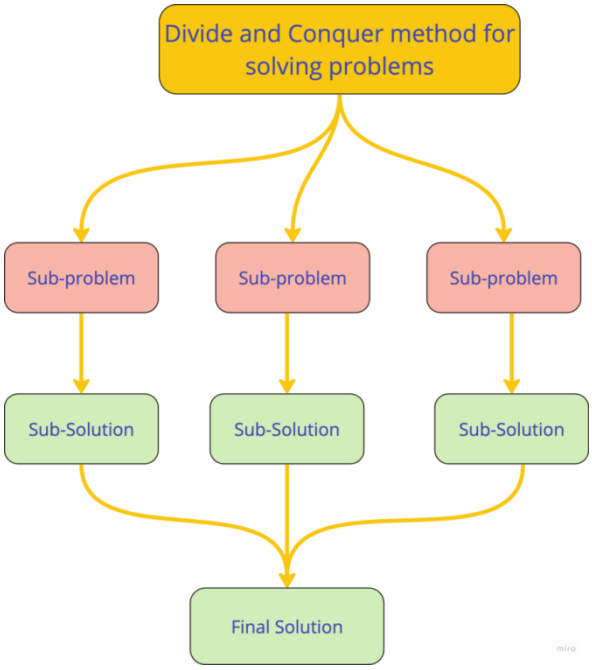
```
import itertools
import string

# Brute-force approach to guess a password
def check_password(password):
    characters = string.printable # contain all the printable characters
    for guess in itertools.product(characters, repeat=3):
      guess = ''.join(guess)  # Convert tuple to string
      print(f"Trying password: {guess}")
      if guess == password:
        print(f"Password found: {guess}")
        return

# Example usage
password = "CaT"  # Set the correct password
check_password(password)
```

Divide and Conquer Algorithm

Divide and Conquer Algorithm is a problem-solving technique used to solve problems by dividing the main problem into sub-problems, solving them individually and then merging them to find solution to the original problem.



Working of Divide and Conquer Algorithm

The **Divide and Conquer** algorithmic approach consists of three main steps: **Divide**, **Conquer**, and **Merge**.

1. Divide

- **Goal**: Break down the original problem into smaller, more manageable subproblems.
- Each subproblem represents a part of the overall problem.
- The division continues recursively until the subproblems are simple enough to solve directly.

---

2. Conquer

- **Goal**: Solve each subproblem individually.
- If a subproblem is small enough (reaching a **base case**), it is solved directly without further recursion.
- Each subproblem is processed independently to find its solution.

---

3. Merge

- **Goal**: Combine the solutions of the subproblems to form the solution to the original problem.
- Once the subproblems are solved, their solutions are recursively merged to resolve the overall problem efficiently.

Characteristics of Divide and Conquer Algorithm

1. **Dividing the Problem**
    - The problem is recursively broken down into smaller subproblems until they become simple enough to solve directly.
2. **Independence of Subproblems**
    - Each subproblem is independent, meaning the solution of one subproblem does not rely on another.
    - This allows for parallel or concurrent execution of subproblems, improving efficiency.
3. **Conquering Each Subproblem**
    - After dividing the problem, each subproblem is solved individually, often applying the divide and conquer strategy recursively.
4. **Combining Solutions**
    - The solutions to the subproblems are merged to form the final solution.
    - This merging step is designed to be efficient and seamless.

Examples of Divide and Conquer Algorithm:

1. Finding the Maximum Element in an Array Using Divide and Conquer

The **Divide and Conquer** algorithm can efficiently find the maximum element in an array by following these steps:

• Divide:

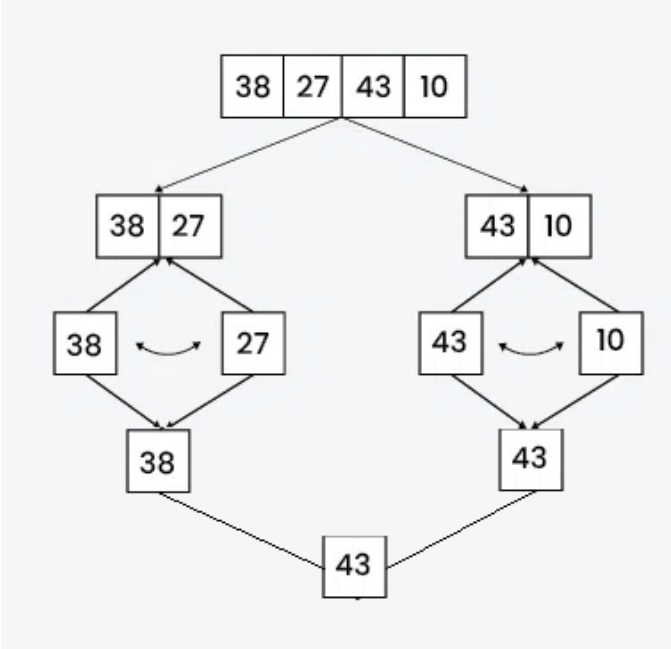
- Split the array into two halves.
- Recursively divide each half until subarrays of size 1 are reached.

• Conquer:

- Once the subarrays have only one element, return that element as the maximum of that subarray.

• Merge:

- Compare the maximums of the two halves and return the larger value as the maximum for the combined array.



Python Program:

```
def find_max(arr, left, right):
    # Base case: when there's only one element
    if left == right:
        return arr[left]

    # Find the middle index
    mid = (left + right) // 2

    # Recursively find the maximum in the left and right halves
    max_left = find_max(arr, left, mid)
    max_right = find_max(arr, mid + 1, right)

    # Return the maximum of the two halves
    return max(max_left, max_right)
```

```
arr = [12, 34, 67, 89, 23, 45, 78]

result = find_max(arr, 0, len(arr) - 1)

print("The maximum element is:", result)
```

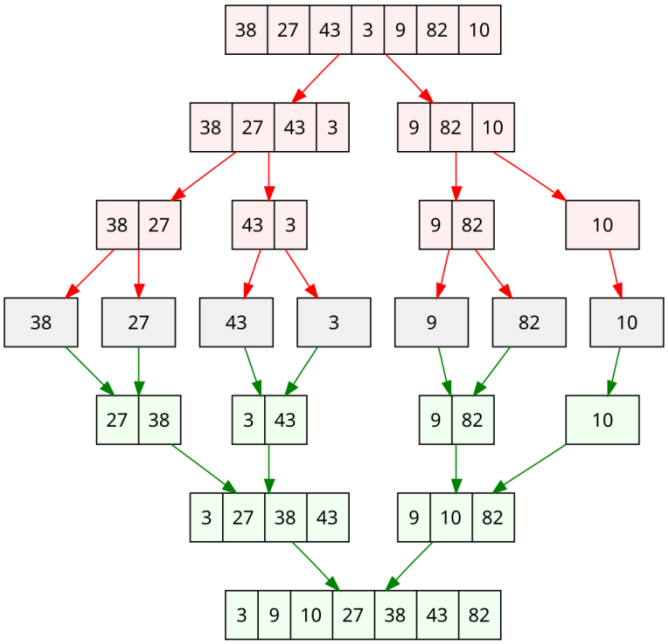
Merge Sort

Ref: Notes/Video : Module 3 – Part – 8

Merge Sort is a classic example of problem decomposition. It uses a divide-and-conquer approach to sort an array.

1. **Divide:** The array is divided into two halves recursively until each subarray has one element.
2. **Sort:** Each sub-array is sorted individually.
3. **Merge:** The sorted sub-arrays are merged to produce a single sorted array.

Merge Sort Example:



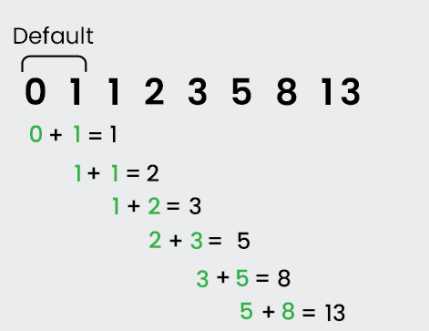
Algorithm Outline:

- **Input:** An unsorted array `arr` of size `n`.
- **Process:**
  - Divide `arr` into two halves `left` and `right`.
  - Recursively sort `left` and `right`.
  - Merge `left` and `right` into a sorted array.
- **Output:** A sorted array.

Dynamic Programming Approach

1. Breakdown Complex problems to sub problems
2. Find the optimal solutions to the sub problems
3. Store the results of sub problems
4. Reuse the results of sub problems to avoid repeated calculations
5. Finally find the result of complex problem

Consider a program to generate Fibonacci series up to n terms using recursive method

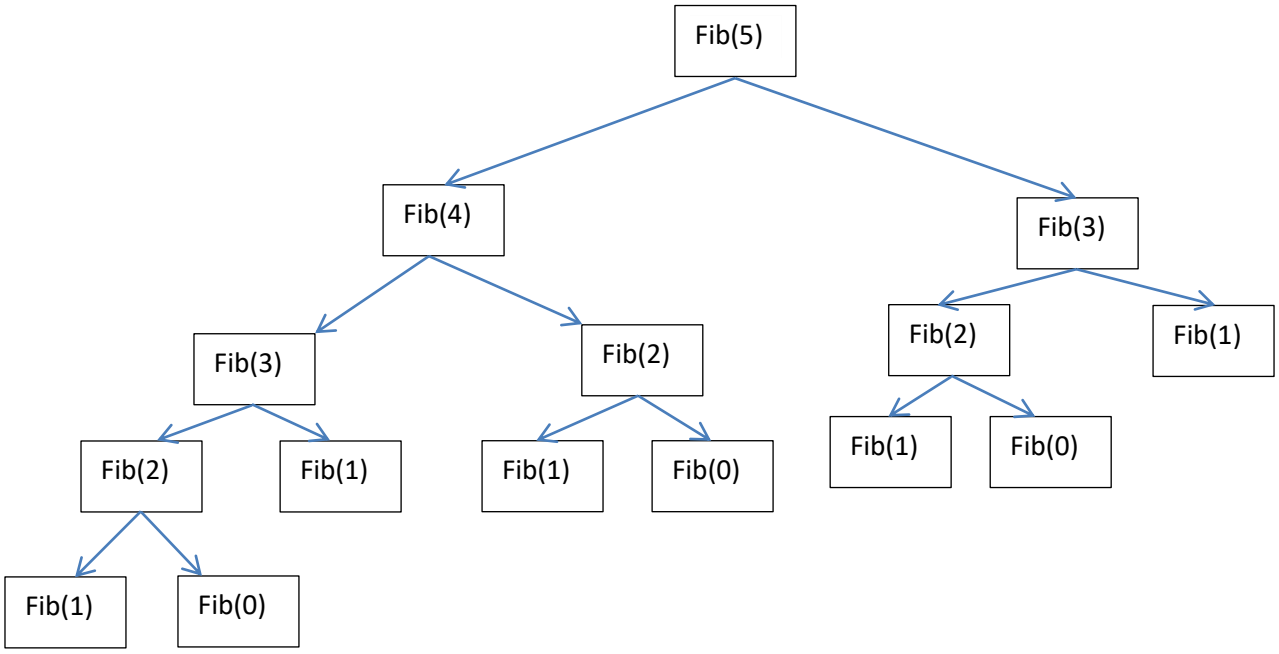



```
def fib(n):
   if n <= 1:
       return n
   else:
       return(fib(n-1) + fib(n-2))



n = int(input("Enter n :"))
if n <= 0:
   print("Plese enter a positive integer")
else:
   print("Fibonacci sequence:")
for i in range(n):
   print(fib(i),end = " ")
```

```
Consider n = 5 ; We know fib(0) = 0 and fib(1) = 1
```



Challenges with Recursive Approach

Even though the recursive approach is intuitive, it often leads to increased time complexity. This is because the same subproblems are repeatedly calculated, resulting in redundant computations and inefficient performance.

Approaches in Dynamic Programming

Dynamic Programming can be implemented using two main approaches:

---

1. Memoization (Top-Down Approach)

- **Definition**:
  Memoization solves the problem recursively while storing the results of subproblems in a table (often a dictionary or array).
- **Process**:
  - When a subproblem is solved, its result is saved.
  - If the same subproblem is encountered again, the solution is retrieved from the table instead of being recalculated.

---

2. Tabulation (Bottom-Up Approach)

- **Definition**:
  Tabulation solves the problem iteratively by filling a table (usually an array) in a bottom-up manner.
- **Process**:
  - Starts by solving the smallest subproblems.
  - Uses their solutions to iteratively build solutions for larger subproblems.
- **Advantages**:
  - Eliminates recursion overhead.
  - Suitable for problems where all subproblems need to be solved.

Solution using Tabulation :

```
def fib(n):

    array = [0] * (n)
    array[0] = 0  # First Fibonacci number
    if n > 1:
        array[1] = 1  # Second Fibonacci number

    for i in range(2, n):
        array[i] = array[i - 1] + array[i - 2]

    return array



n = int(input("Enter the number of terms: "))
if n <= 0:
    print("Please enter a positive integer")
else:
    print("Fibonacci sequence:")
    print(fib(n))
```

Compare Recursion and Dynamic Programming

Features	Recursion	Dynamic Programming
Definition	By breaking a difficulty down into smaller problems of the same problem, a function calls itself to solve the problem until a specific condition is met.	It is a technique by breaks them into smaller problems and stores the results of these subproblems to avoid repeated calculations.
Approach	Recursion frequently employs a top-down method in which the primary problem is broken down into more manageable subproblems.	Using a bottom-up methodology, dynamic programming starts by resolving the smallest subproblems before moving on to the primary issue.
Base Case	To avoid infinite loops, it is necessary to have a base case (termination condition) that stops the recursion when a certain condition is satisfied.	Although dynamic programming also needs a base case, it focuses mostly on iteratively addressing subproblems.
Performance	Recursion might be slower due to the overhead of function calls and redundant calculations.	Dynamic programming is often faster due to optimized subproblem solving and memoization.
Memory Usage	It does not require extra memory, only requires stack space	Dynamic programming require additional memory to record intermediate results.

Features	Recursion	Dynamic Programming
Time Complexity	**Higher, often exponential for problems with overlapping subproblems.**	**Lower**
Performance	Slower for problems with overlapping subproblems.	Faster due to elimination of redundant calculations.

Greedy Algorithm for Problem Solving

A ***Greedy Algorithm*** is a problem-solving approach that builds up a solution piece by piece, always choosing the option that looks best at every step. It focuses on making a sequence of locally optimal choices, with the hope that these decisions will lead to a globally optimal solution.

Example: Traveling from Thiruvananthapuram to Ernakulam Using the Greedy Approach

Let's consider the task of traveling from Thiruvananthapuram to Ernakulam with multiple available modes of transport: bike, car, bus, train, airplane, or even walking. The goal is to select the mode of transport based on a set of constraints or objectives:

1. **Minimizing Travel Time**
2. **Minimizing Cost**
3. **Satisfying Practical Constraints**

Step 1: Apply the Greedy Approach

At each step, the greedy algorithm makes the *locally optimal choice* by filtering options based on the immediate constraint.

Stage 1: Minimize Travel Time

To reach the destination quickly, we first prioritize speed.

- **Available Options**:
  - **Bike**: 6 hours
  - **Car**: 4.5 hours
  - **Bus**: 5 hours
  - **Train**: 3.5 hours
  - **Airplane**: 1 hour
  - **Walking**: 40 hours

Greedy Decision:
Select modes that satisfy the time constraint of reaching as quickly as possible. The airplane (1 hour) is the fastest, followed by the train (3.5 hours).

Filter Outcome:
Airplane, Train.

Stage 2: Minimize Cost (Economical Constraint)

Among the remaining options, we now focus on minimizing cost.

- **Cost of Airplane**: ₹3,500
- **Cost of Train**: ₹500

Greedy Decision:
Select the train as it is cheaper and satisfies the economical constraint.

Filter Outcome:
Train.

Stage 3: Practical Constraints

Lastly, consider practical constraints like availability of tickets, comfort, or personal preferences. If the train has tickets available and meets other needs, it will be chosen.

Final Decision

The algorithm selects the **train** as the final mode of transport based on:

1. Minimizing time (after filtering slower options).
2. Minimizing cost.
3. Satisfying practical constraints.

Greedy Approach Explanation

- **Locally Optimal Choices**: At each step, the algorithm optimizes for the immediate constraint (time, cost, practicality) without revisiting earlier decisions.
- **Global Solution**: The sequence of locally optimal decisions leads to an efficient and practical final solution.

Example: Coin Changing Problem

The Coin Changing Problem aims to find the minimum number of coins required to make a specified amount using valid Indian Rupee coins. The greedy algorithm repeatedly selects the largest denomination that fits into the remaining amount.

Valid Indian Coin Denominations:

₹1, ₹2, ₹5, ₹10

Example:

To make ₹18 using the greedy algorithm:

1. **Start with ₹10: Take one ₹10 coin (₹18 - ₹10 = ₹8 left).**
2. **Next, ₹5: Take one ₹5 coin (₹8 - ₹5 = ₹3 left).**
3. **Next, ₹2: Take one ₹2 coin (₹3 - ₹2 = ₹1 left).**
4. **Finally, ₹1: Take one ₹1 coin (₹1 - ₹1 = ₹0 left).**

Total coins used: 4 (1×₹10, 1×₹5, 1×₹2, 1×₹1).

Motivations for the Greedy Approach

The Greedy Approach is an effective problem-solving strategy due to several key motivations:

1. **Simplicity and Ease of Implementation**
   - **Straightforward Logic:** Makes optimal local choices, simplifying understanding and implementation.
   - **Minimal Requirements:** Requires less complex data structures
2. **Efficiency in Time and Space**
   - **Fast Execution:** Suitable for large inputs.
   - **Low Memory Usage:** Uses minimal memory by avoiding extensive intermediate storage.
3. **Optimal Solutions for Specific Problems**
   - **Greedy-Choice Property:** Local optimal choices lead to a global optimum.
   - **Optimal Substructure:** Global optimal solutions can be built from optimal subproblem solutions.
4. **Real-World Applicability**
   - **Practical Applications:** Used in scheduling, network routing, and resource allocation.
   - **Quick, Near-Optimal Solutions:** Offers efficient solutions when exact results aren't necessary.

Characteristics of Greedy Algorithms

1. **Local Optimization**
   - Makes the best possible choice at each step using only current state information.
2. **Irrevocable Decisions**
   - Choices are final; no backtracking or revision of earlier decisions.
3. **Problem-Specific Heuristics**
   - Relies on heuristics tailored to the problem's properties for decision-making.
4. **Optimality**

- Guarantees optimal solutions for problems like coin change, Huffman coding, and Kruskal's algorithm, but not universally applicable.

5. **Efficiency**
   - High efficiency in time and space due to reliance on local information and limited exploration of solutions.

Problem (Task Completion Problem)
Given an array of positive integers each indicating the completion time for a task, find the maximum number of tasks that can be completed in the limited amount of time that you have.

Consider the example usage with `completion_times = [2, 3, 1, 4, 6]` **and** `available_time = 8`

Steps:

1. **Sort Tasks**: Arrange tasks by completion time (ascending).
2. **Iterate & Track**: Add tasks sequentially as long as the total time doesn't exceed the limit. Update the task count accordingly.

• After sorting: [1, 2, 3, 4, 6]

• Iterating:

– Add task with time 1: total_time = 1, task_count = 1

– Add task with time 2: total_time = 3, task_count = 2

– Add task with time 3: total_time = 6, task_count = 3

– Next task with time 4 would exceed available_time, so the loop breaks.

The maximum number of tasks that can be completed in 8 units of time is 3.

Program:

```
def max_tasks(completion_times, available_time):
    completion_times.sort()  # Step 1: Sort tasks by completion times
    total_time = 0
    task_count = 0

    # Step 2: Iterate through the sorted list of tasks
    for time in completion_times:
        if total_time + time <= available_time:
            total_time += time
            task_count += 1
        else:
            break
    return task_count

# Example usage
completion_times = [2, 3, 1, 4, 6]
available_time = 8
print(f"Maximum number of tasks that can be completed:
{max_tasks(completion_times, available_time)}")
```

Comparison between Dynamic programming and Greedy Approach

Feature	Greedy Method	Dynamic programming
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
Memorization	It is more efficient in terms of memory as it never look back or revise previous choices	It requires Dynamic Programming table for Memorization and it increases its memory complexity.
Time complexity	Greedy methods are generally faster.	Dynamic Programming is generally slower.

Advantages and Disadvantages

- **Advantages**
  - **Easy to implement**: Greedy algorithms are relatively easy to understand and implement.
  - **Time complexity**: Greedy algorithms usually have a smaller time complexity.
  - **Optimization**: Greedy algorithms can be used for optimization or to find solutions that are close to optimal for hard problems.
- **Disadvantages**
  - **Not guaranteed to find the best solution**: Greedy algorithms may not find the best solution because they don't consider all the data.
  - **Local optima**: Greedy algorithms may get stuck in local optima and fail to find the global optimum.
  - **Lack of backtracking**: Once a decision is made, it cannot be undone.
  - **Dependence on problem structure**: Greedy algorithms may not work well for problems that don't fit the greedy paradigm.
  - **Lack of rigorous proof**: Greedy algorithms often lack a rigorous proof of correctness

Problem Details

Mr. Chottu has a bag with a capacity of 15 kg, and he plans to fill it with objects to maximize his profit as he heads to the market for a sale. He has the following items with their respective weights and profits:

Objects	Weight	Profit
A	2	10
B	3	5
C	5	15
D	7	7
E	1	6
F	4	18
G	1	3

Types of Knapsack Problems

1. **0/1 Knapsack Problem**:
   - Items cannot be divided; you either take the whole item or leave it.
   - Example: Carrying specific tools in a toolbox.
2. **Fractional Knapsack Problem**:
   - Items can be divided into fractions, allowing you to take portions of items.
   - Example: Packing grains in a bag where partial quantities are possible.

Objective: (Here we are following Fractional Knapsack Approach)

Maximize profit by optimizing the selection of objects, considering the profit-to-weight (P/W) ratio.

Steps:

1. **Calculate P/W ratio for each object.**
   P/W=Profit ÷Weight
2. **Sort objects in descending order of P/W ratio.**
3. **Iteratively add objects to the knapsack:**
   - If the object can fit entirely in the knapsack, take it.
   - If the object cannot fit entirely, take the fractional part.
4. **Stop when the knapsack is full.**

Table of Calculations:

Object	Profit (P)	Weight (W)	P/W
1	10	2	5.0
2	5	3	1.67
3	15	5	3.0
4	7	7	1.0
5	6	1	6.0
6	18	4	4.5
7	3	1	3.0

Sorted by P/W (Descending):

Object	Profit (P)	Weight (W)	P/W
5	6	1	6
1	10	2	5
6	18	4	4.5
3	15	5	3
7	3	1	3
2	5	3	1.67
4	7	7	1

Execution:

Start with the highest P/W:

1. Take Object 5 (6 profit, 1 kg):
   Remaining capacity: 15−1=14
   Total profit: 6
2. Take Object 1 (10 profit, 2 kg):
   Remaining capacity: 14−2=12
   Total profit: 6+10=16
3. Take Object 6 (18 profit, 4 kg):
   Remaining capacity: 12−4=8
   Total profit: 16+18=34
4. Take Object 3 (15 profit, 5 kg):
   Remaining capacity: 8−5=3
   Total profit: 34+15=49
5. Take Object 7 (3 profit, 1 kg):
   Remaining capacity: 3−1=2
   Total profit: 49+3 = 52
6. Take fraction of Object 2 :
   Remaining capacity: 2−2 = **0**
   Total profit: 52+(2*1.67) = 55.34

Python's `random` module provides various functions to generate random numbers and perform random operations. Here's an overview of the most commonly used functions:

1. Basic Random Number Generators

- **`random.random()`**
  - Returns a random float number between 000 (inclusive) and 111 (exclusive).
  - Example:

```
import random
print(random.random())  # e.g., 0.7234
```

- **`random.uniform(a, b)`**
  - Returns a random float between aaa and bbb (inclusive).
  - Example:

```
print(random.uniform(1, 10))  # e.g., 7.3521
```

- **`random.randint(a, b)`**
  - Returns a random integer between aaa and bbb (both inclusive).
  - Example:

```
print(random.randint(1, 10))  # e.g., 4
```

2. Working with Sequences

- **`random.choice(sequence)`**
  - Returns a random element from a non-empty sequence (like a list, tuple, or string).
  - Example:

```
items = ['apple', 'banana', 'cherry']
print(random.choice(items))  # e.g., 'banana'
```

- **`random.choices(sequence, k=n)`**
  - Returns a list of `n` random elements (with replacement).
  - Example:

```
print(random.choices(items, k=2))  # e.g., ['apple', 'cherry']
```

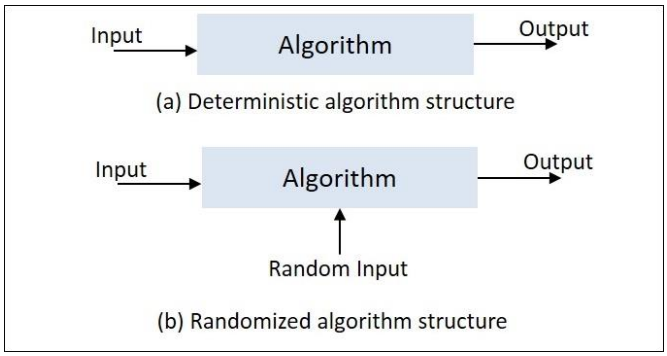
- **`random.shuffle(sequence)`**
  - Shuffles the sequence in place.
  - Example:

```
nums = [1, 2, 3, 4, 5]
random.shuffle(nums)
print(nums)  # e.g., [3, 1, 5, 2, 4]
```

Randomized Approach to Problem Solving

Introduction

The **randomized approach** in computational problem-solving leverages randomness to simulate and analyze complex problems. Unlike deterministic methods, it provides probabilistic solutions that can simplify problem complexity and offer practical insights.



Calculating the area of a unit circle using a randomized approach involves **Monte Carlo simulation**, a technique often used in numerical methods to estimate values through random sampling.

Steps to Calculate the Area of a Unit Circle Using Monte Carlo Simulation:

1. **Understand the Problem**:
   - The unit circle has a radius of 1, and its area is given by $\pi r^2 = \pi$ (since r=1).
   - We want to estimate this area.
2. **Define the Bounding Square**:
   - The unit circle is inscribed in a square with side length 2 (since the diameter of the circle is 2).
   - The square spans from $(-1,-1)$ to $(1,1)$ on a Cartesian plane.
3. **Random Sampling**:
   - Randomly generate points (x,y) within the bounding square.
   - Each x and y value is chosen uniformly between $-1$and 1
4. **Check if Points Lie Inside the Circle**:
   - A point (x,y) lies inside the circle if $x^2+y^2 \leq 1$
5. **Calculate the Ratio**:
   - Let N_{circle} be the number of points that lie inside the circle.
   - Let N_{total} be the total number of points generated.
   - The ratio N_{circle} / N_{total} approximates the ratio of the area of the circle to the area of the square.
   - Since the area of the square is 4, the area of the circle is approximately: Area of Circle $\approx 4\times(N_{circle} / N_{total})$.
6. **Iterate for Accuracy**:
   - The more points you sample, the better the approximation.

Program

```
import random

num_points = 100000

inside_circle = 0

for _ in range(num_points):

    x = random.uniform(-1,1)

    y = random.uniform(-1,1)

    if x**2 + y**2 <= 1:

        inside_circle = inside_circle + 1


area_of_circle = 4 * (inside_circle/num_points)

print(f"Estimated area : {area_of_circle}")
```

Example : **Random Coin Flips**

- **Problem**: Estimate the probability distribution of heads in a series of coin flips.
- **Approach**:
  - Simulate repeated coin flips.
  - Record outcomes and analyze the distribution of heads.
- **Application**: Empirical analysis of probabilistic events.

Key Concepts and Examples

1. **Monte Carlo Simulation (Estimating Circle Area)**
   - **Problem**: Estimate the area of a circle inscribed in a square.
   - **Approach**:
     - Randomly place points within the square.
     - Calculate the proportion of points that fall inside the circle.
     - Use this ratio to estimate the circle's area relative to the square.
   - **Application**: Solves geometric problems via probabilistic modeling.
2. **Birthday Problem**
   - **Problem**: Find the probability that at least two people in a group share the same birthday.

- **Approach**:
  - Simulate birthdays for a group multiple times.
  - Record how often at least two birthdays match.
  - Estimate the probability based on these simulations.
- **Application**: Simplifies combinatorial probability analysis.

3. **Random Walk on a Grid**
   - **Problem**: Determine the expected return time to the starting point for a person taking random steps on a grid.
   - **Approach**:
     - Simulate random walks.
     - Calculate average return time from multiple trials.
   - **Application**: Models stochastic processes and dynamics.
4. **Random Coin Flips**
   - **Problem**: Estimate the probability distribution of heads in a series of coin flips.
   - **Approach**:
     - Simulate repeated coin flips.
     - Record outcomes and analyze the distribution of heads.
   - **Application**: Empirical analysis of probabilistic events.

Coin Toss Experiment

Tossing a fair coin once

Let us say we have a fair coin and toss the coin 'n' times. We will observe either a head or a tail in each flip. We can code 1 for head and 0 for tail.

```
import random

head=0

tail=0

n=int(input("Enter the number of coin tosses you want to perform
: "))

for _ in range(n):

    if random.randint(0,1)==1:
```

```
        head=head+1

    else:

        tail=tail+1

print("Number of heads = ",head)

print("Number of tails = ",tail)
```

Q. Toss the coin N=100,500,1000,5000 and 500000 times and compute the probability (p) of head in each case.

```
import random
n = [100,500,1000,5000,500000]
prob=[]
count=0
while(count<len(n)):
    head=0
    tail=0
    for i in range(n[count]):
        if random.randint(0,1)==0:
            head=head+1
        else:
            tail=tail+1
    p=head/(head+tail)
    prob.append(p)
    count=count+1
print(prob)
```

Motivations for Randomized Approach

1. **Complexity Reduction**
   - Simplifies problems by introducing probabilistic decisions.
   - Example: Optimizing screening station placements by sampling.
2. **Versatility**
   - Adaptable across domains like optimization and simulations.
3. **Performance**
   - Enhances efficiency in large datasets.

Characteristics of Randomized Approach

1. **Probabilistic Choices**
    - Decisions are based on random sampling, leading to variable but statistically predictable outcomes.
2. **Efficiency**
    - Sacrifices deterministic guarantees for probabilistic correctness.
3. **Average-Case Complexity**
    - Performance is evaluated over multiple iterations, focusing on typical behavior rather than worst-case scenarios.

Comparison: Randomized vs. Deterministic Methods

- **Randomized**:
    - Incorporates chance.
    - Efficient for large, complex, or variable datasets.
    - Example: Estimating customer satisfaction via random sampling.
- **Deterministic**:
    - Yields exact, repeatable results.
    - Suitable for precision-critical problems.
    - Example: Exact cost calculation in shopping.