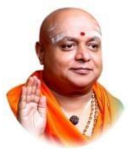


|| Jai Sri Gurudev ||

Sri Adichunchanagiri Shikshana Trust (R)

SJB Institute of Technology

BGS Health and Education City, Dr. Vishnuvardhana Road, Kengeri, Bengaluru-560060

Approved by AICTE, New Delhi.

Autonomous Institute affiliated to Visvesvaraya Technological University, Belagavi

Accredited by NAAC with 'A+'grade, Certified by ISO 9001 - 2015

Recognized by UGC, New Delhi with 2(f) & 12 (B)

Department of Mathematics

Third Semester Python Lab manual for a Computer Science and Engineering Stream

Course Title & Code:
Discrete Mathematics and Graph Theory(23CSI301)

Academic Year: 2024-25

Name	:	______________________________
USN	:	______________________________
Section	:	______________________________

SJB INTITUTE OF TECHNOLOGY

Institution's Vision

To become a recognized technical education center with a global perspective.

Institution's Mission

To provide learning opportunities that foster students ethical values, intelligent development in science & technology and social responsibility so that they become sensible and contributing members of the society.

Department of Mathematics

Department Vision

Is envisaging to become world's foremost technical school fostering intellectual excellence.

Department Mission

To develop technically challenging leaders and entrepreneurs for facilitating and enhancing the global competitiveness of organization through excellence in education, research and training.

PROGRAM OUTCOMES

Engineering graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

Program Educational Objectives(PEOs)

Program Specific Outcomes(PSOs)

General Instructions for the Laboratory

Do’s

- It is mandatory for all the students to attend all practical classes &complete the experiments as per syllabus.
- Students should strictly follow the lab timings, dress code with Apron & ID cards. Should maintain a neat observation book.
- Study the theory and logic before executing the program.
- Submit the completed lab records of executed programs and update the index book in every lab session.
- Should prepare for viva questions regularly. Handle the computer systems carefully.
- Maintain discipline and silence in the lab.

Don’ts

- Should not take Bags and Mobile phones into the Laboratory.
- Do not wear footwear inside the Laboratory.
- Systems &Components should be handled carefully failing to which penalty will be imposed.
- Do not switch off the system abruptly.
- Should not chew gum or eat in the lab.

COURSE OUTCOMES (With PO & PSO Mapping)

Semester	3rd	**Course title & Course code**	Discrete Mathematics and Graph Theory(23CSI301)

Course Outcomes(COs)		**PO & PSO**
CO1	Develop computational problem-solving skills by implementing Python programs to assess logical consistency and verify whether a given proposition is a tautology.	
CO2	Deploy Python to apply and verify mathematical concepts, including induction and combinatorial methods.	
CO3	Implement Python programs for function properties, equivalence relations, Fibonacci sequences, and partial orders.	
CO4	Create Python programs for graph coloring, TSP, edge-disjoint paths, and vertex cuts.	
CO5	Not required	

Correlation Matrix

CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	1		1							1			
CO2	3	2	1		1							1			
CO3	3	2	1		1							1			
CO4	3	2	1		1							1			
CO5	3	2	1		1							1			

Note: **Correlation levels: 1: Slight (Low) 2: Moderate (Medium) 3: Substantial (High), "-" → No correlation**

Discrete Mathematics and Graph Theory

Choice Based Credit System (CBCS) and Outcome Based Education (OBE)

Laboratory Code	**23CSI301**	**CIE Marks**	**50**
Teaching Hours/Week (L:T:P: S)	**0:0:2:0**	**SEE Marks**	**50**
Credits		**Exam Hours**	**03**

Course Learning Objectives:

1. Apply Python to evaluate logical connectives and determine tautologies.
2. Utilize Python to compute sums of sequences and perform permutations and combinations.
3. Develop Python functions to analyze function properties, such as one-to-one and onto, and assess equivalence relations.
4. Implement both iterative and recursive approaches to the Fibonacci sequence using Python, and verify partial orders and graph coloring.
5. Solve advanced graph problems with Python, including the Traveling Salesman Problem (TSP), edge-disjoint paths, and minimum vertex cuts using Menger's theorem.

Lab No.	**Programs**
1	Write a program on logical connectives (AND, OR, NOT, XOR).
2	Check whether the given proposition is a tautology or not.
3	Compute the sum of first n odd numbers using mathematical induction.
4	Calculation of Permutation and combination.
5	Implement functions to check whether a given function is one-to-one and onto.
6	Check whether the relation is equivalence or not.
7	Implement the Fibonacci sequence using both an iterative approach and a recursive approach.
8	Write a program to verify a given relation forms a partial order or not.
9	Write a program on assign colors to the vertices of a graph, no two adjacent vertices share the same color.
10	Implement the Traveling Salesman Problem (TSP) using a Hamilton path approach to find the shortest Hamilton Path in a weighted graph.
11	Write a program to find the maximum number of edge-disjoint paths between two vertices. Use the Edmonds-Karp algorithm, an implementation of the Ford Fulkerson method for computing the maximum flow in a flow network.
12	Using Menger's theorem, find the minimum vertex cut between source and target.

CONTENTS

Preface

This manual introduces Discrete Mathematics and Graph Theory with a focus on Python programming for Computer Science Engineering stream students. This guide aims to provide a solid foundation in these crucial areas while demonstrating practical applications using Python.

Discrete mathematics underpins many computer science concepts, from algorithms to data structures. Graph theory, a key branch, helps in solving problems related to networks, optimization, and more.

In this manual, you'll find clear explanations of core topics such as set theory, combinatorics, and graph algorithms. Each concept is illustrated with Python code with practical implementation.

The manual includes exercises to reinforce learning and real-world applications to show the relevance of these concepts in computer science.

Materials and Setup

Software Requirements:

- Jupyter Notebook: An interactive environment for writing and running Python code, ideal for experiments and visualizations.
- PyCharm: A robust Integrated Development Environment(IDE) for Python development with advanced features for code management and debugging.
- VS Code: A versatile editor with support for Python, including useful extensions for enhanced development.
- Anaconda: A distribution that includes Python, Jupyter Notebook, and commonly used scientific libraries, simplifying package management and deployment.
- Python 3.x: The programming language version required for compatibility with libraries and code examples.
- NumPy and Pandas: Libraries for numerical computing and data manipulation, often used in conjunction with Jupyter Notebook.
- Matplotlib and Seaborn: Libraries for data visualization, useful for graphical representation of results and analysis.

Hardware Requirements:

- Computer with Sufficient RAM: At least 8GB of RAM is recommended for smooth performance, especially for handling large datasets and running complex algorithms.
- Processor: A multi-core processor (e.g., Intel i5/i7 or Advanced Micro Devices (AMD) Ryzen) for efficient code execution and handling multiple tasks.
- Storage Space: Adequate disk space for installing software, libraries, and storing data files, ideally with an SSD for faster performance.

Instructions and method of evaluation

1. In each lab student have to show the record of previous lab.
2. Each lab will be evaluated for 15 marks and finally average will be taken for 15marks.
3. Viva-voce questions shall be asked in labs and attendance also can be considered foreveryday lab evaluation.
4. Tests shall be considered for 5 marks and final lab assessment is for 20 marks.
5. Student has to score minimum 8 marks out of 20 to pass lab component.

LAB 1: Write a program on logical connectives (AND, OR, NOT, XOR).

1.1 Objectives:

Use python

- to perform AND, OR, NOT, XOR operations.
- Evaluate the connectives with all possible combinations of boolean values.
- to verify the correctness of the logical operations.

1.2 Write a program on logic gates (AND, OR, NOT):

Logical connectives are fundamental in programming and are used to create complex logical expressions by combining simpler ones. The primary logical connectives are AND (**and**), OR (**or**), and NOT (**not**). Below are a few Python programs demonstrating the use of these logical connectives.

1. **Logical AND**

The AND operator returns True if both operands are True.

```
def logical_and(a, b):
    return a and b

# Truth Table for AND
print("A     B     A  AND B")
for a in [True, False]:
    for b in [True, False]:
        print(f"{a}  {b}  {logical_and(a, b)}")
```

OUTPUT:

```
A          B        A AND B
True      True       True
True      False      False
False     True       False
False     False      False
```

2. **Logical OR**

The OR operator returns True if at least one of the operands is True.

```
def logical_or(a, b):
    return a or b

# Truth Table for OR
print("A     B     A OR B")
for a in [True, False]:
    for b in [True, False]:
        print(f"{a}  {b}  {logical_or(a, b)}")
```

OUTPUT:

```
  A         B         A OR B
True     True       True
True     False      True
False    True       True
False    False      False
```

3. Logical NOT

The NOT operator returns True if the operand is False and vice versa.

```
def logical_not(a):
    return not a

# Truth Table for NOT
print("A     NOT A")
for a in [True, False]:
    print(f"{a}  {logical_not(a)}")
```

OUTPUT:

```
A         NOT A
True       False
False      True
```

4. Logical XOR (^)

The XOR operator returns True if exactly one of the operands is True.

```
def logical_xor(a, b):
    return a ^ b

# Truth Table for XOR
print("A     B     A XOR B")
for a in [True, False]:
    for b in [True, False]:
        print(f"{a}  {b}  {logical_xor(a, b)}")
```

OUTPUT:

```
A           B       A XOR B
True      True     False
True      False    True
False     True     True
False     False    False
```

5. Combining Logical Connectives

Write a Python program that prints a truth table for the logical expression (A AND B) OR (NOT C) AND (A XOR C).

Let's combine these connectives in a complex logical expression:

```
def complex_expression(a, b, c):
    return (a and b) or (not c) and (a ^ c)

# Truth Table for the complex expression
print("A     B     C     (A AND B) OR (NOT C) AND (A XOR C)")
for a in [True, False]:
    for b in [True, False]:
        for c in [True, False]:
            print(f"{a}  {b}  {c}  {complex_expression(a, b, c)}")
```

```
A                  B                C        (A AND B) OR (NOT C) AND (A XOR C)
True              True             True                        True
True               True            False                       True
True               False           True                         False
True              False            False                       True
False              True            True                         False
False             True             False                       True
False              False           True                         False
 False             False           False                       True
```

6. Logical Connectives in Real-world Scenarios- Check the eligibility for a Loan:

A person is eligible for a loan if they have a good credit score, a stable job, and no outstanding debts.

```
def check_loan_eligibility(credit_score, has_stable_job, has_outstanding_debts):
    if credit_score >= 700 and has_stable_job and not has_outstanding_debts:
        return "Eligible for loan"
    else:
        return "Not eligible for loan"

# Example usage
credit_score = 650
has_stable_job = True
has_outstanding_debts = False

result = check_loan_eligibility(credit_score, has_stable_job, has_outstanding_debts)
print(result)
```

OUTPUT:

```
Not eligible for loan
```

1.3 Exercise:

1. Write a Python program that prints a truth table for the following logical expression

   i) (A and B) or (not C).

   ii) (A or not B) and (not C).

   iii) (A or B) and (C or not D) and not (A and C).

Viva Questions and Answers

1. **What are logical connectives?**

   Answer: Logical connectives are operations used in logic to combine or modify boolean values (True or False). Common logical connectives include AND, OR, NOT, and XOR.
2. **Explain the AND operation.**

   Answer: The AND operation returns True if and only if both operands are True. If either operand is False, the result is False.
3. **Explain the OR operation.**

   Answer: The OR operation returns True if at least one of the operands is True. It only returns False if both operands are False.
4. **What is the purpose of the NOT operator?**

   Answer: The NOT operator inverts the boolean value of its operand. If the operand is True, it becomes False and vice versa.
5. **What is XOR and how does it work?**

   Answer: XOR, or Exclusive OR, returns True only if exactly one of the operands is True. If both operands are the same (both True or both False), XOR returns False.
6. **What is the difference between XOR and OR?**

   Answer: OR returns True if one or both operands are True, while XOR returns True only if one operand is True and the other is False.
7. **Can logical connectives be used with non-boolean values?**

   Answer: Yes, in many programming languages, logical connectives can be applied to non-boolean values. For example, in Python, any non-zero value is treated as True, and zero or None is treated as False.

8. **How would you implement logical connectives in Python?**

   Answer: You can use the and, or, and not keywords in Python to implement logical connectives, while XOR can be implemented using the ^ operator for integers or a combination of and and not for booleans.

9. **Can you write a function that accepts three boolean inputs and returns the result of applying AND, OR, and XOR to them?**

   Answer: Yes, the function can use the three inputs and apply the and, or, and XOR operations as demonstrated in the program.

10. **How would you handle logical operations on lists or arrays in Python?**

    Answer: In Python, logical operations on lists or arrays can be handled element-wise using list comprehensions, or by using the numpy library which supports vectorized logical operations.

11. **What would happen if you tried to apply the NOT operator to a non-boolean value in Python?**

    Answer: If applied to a non-boolean value, Python treats any non-zero value as True and zero as False, so the NOT operator would invert their boolean equivalent.

12. **What real-life applications use logical connectives extensively?**

    Answer: Logical connectives are used in areas like digital circuit design, conditional statements in programming, decision-making algorithms, and artificial intelligence systems.

LAB 2: Check whether the given proposition is a tautology or not.

2.1 Objectives:

Use python

- to generate and evaluate the truth table.
- To determine if the proposition is a tautology.

2.2 Determine whether the following propositions are tautologies or not:

1. (p or q) and (not p or q).

```
import itertools

def evaluate_proposition(p, q, expression):
    # Evaluate the expression with the given truth values of p and q
    return eval(expression)

def is_tautology(expression):
    # Generate all possible truth values for p and q
    variables = {'p', 'q'}
    truth_values = list(itertools.product([True, False], repeat=len(variables)))

    # Check if the expression is true for all possible truth values
    for values in truth_values:
        p, q = values
        if not evaluate_proposition(p, q, expression):
            return False
    return True

# Example usage
expression = "(p or q) and (not p or q)"  # Example proposition
result = is_tautology(expression)
print("The proposition is a tautology." if result else "The proposition is not a tautology.")
```

OUTPUT:

```
The proposition is not a tautology.
```

2. ¬ (p ∨ q) → (¬ p ∧ ¬ q).

```
from itertools import product

def evaluate_expression(p, q):
    # Evaluate ¬(p ∨ q) → (¬p ∧ ¬q)
    return (not (p or q)) <= (not p and not q)

def is_tautology():
    # Generate all possible truth values for p and q
    truth_values = list(product([True, False], repeat=2))

    # Check if the expression is true for all combinations
    for values in truth_values:
        p, q = values
        if not evaluate_expression(p, q):
            return False
    return True

# Check if the proposition is a tautology and print the result
result = is_tautology()
print(f"The proposition ¬(p ∨ q) → (¬p ∧ ¬q) is {'a tautology' if result else 'not a tautology'}.")
```

OUTPUT:

```
The proposition ¬(p ∨ q) → (¬p ∧ ¬q) is a tautology.
```

2.3 Exercise:

1. Check whether the following proposition is a tautology or not:

   i) $\neg(p \vee \neg q) \rightarrow (\neg p \wedge q)$

   ii) $(p \wedge (q \vee r)) \rightarrow ((p \wedge q) \vee (p \wedge r))$

   iii) $(p \rightarrow q) \wedge (q \rightarrow r) \wedge (p \rightarrow r)$

Viva Questions and Answers

1. What is a proposition in logic?

Answer: A proposition is a declarative statement that can either be True or False, but not both. Propositions are used as the fundamental building blocks in propositional logic.

2. What is a tautology?

Answer: A tautology is a logical statement that is always True, regardless of the truth values of the individual components in the expression. For example, $P \vee \neg P$ (P or not P) is a tautology because it is always True no matter what the truth value of P is.

3. How do you determine if a proposition is a tautology?

Answer: To determine if a proposition is a tautology, you create a truth table that evaluates the proposition for all possible combinations of truth values for the involved variables. If the proposition is True for every possible combination, it is a tautology.

4. What is a truth table?

Answer: A truth table is a mathematical table used to compute the truth values of a logical expression based on the truth values of its variables. It lists all possible combinations of input values and the corresponding output of the logical expression.

5. Can you give an example of a tautology?

Answer: Yes. The expression $(P \wedge Q) \Longrightarrow (P \vee Q)$ is a tautology. It evaluates to True for all possible truth values of P and Q, as shown by constructing a truth table.

6. What is the difference between a tautology and a contradiction?

Answer: A tautology is a logical expression that is always True, whereas a contradiction is a logical expression that is always False. For example, $P \wedge \neg P$ is a contradiction because it is always False, no matter the truth value of PPP.

7. What is logical equivalence?

Answer: Logical equivalence between two propositions occurs when both propositions have the same truth value for every possible combination of truth values for the variables involved. For example, $P \Longrightarrow Q$ is logically equivalent to $\neg P \vee Q$.

8. How would you check for logical equivalence between two propositions?

Answer: To check for logical equivalence, you construct a truth table for both propositions. If the output columns for both propositions are identical for every combination of truth values, then the propositions are logically equivalent.

9. What is the difference between AND, OR, and XOR?

Answer:

i. **AND (∧)**: Returns True if both operands are True.
ii. **OR (∨)**: Returns True if at least one of the operands is True.
iii. **XOR (Exclusive OR)**: Returns True if exactly one of the operands is True, but not both.

10. What are some real-world applications of propositional logic?

Answer: Propositional logic is used in areas like digital circuit design, software engineering (particularly in if-else conditions and decision-making algorithms), artificial intelligence, formal verification of systems, and automated reasoning systems.

LAB 3: Compute the sum of first n odd numbers using mathematical induction.

3.1 Objectives:

Use python

- to compute the sum of the first n odd and even numbers.
- to establish the validity of the sum of the first n natural numbers using both the basic step and an inductive step.

3.2 Mathematical induction:

It is a fundamental technique in mathematics used to prove statements about natural numbers. It's particularly useful for proving properties or formulas that hold for all integers greater than or equal to a certain number.

Steps of Mathematical Induction

Mathematical induction generally involves two main steps:

1. **Basic Step**: Verify that the statement holds for the initial value (often $n=0$ or $n=1$).
2. **Inductive Step**: Assume the statement holds for an arbitrary positive integer k (this is called the **inductive hypothesis**), and then prove that it must also hold for $k+1$.

If both steps are successfully proven, then the statement is true for all integers greater than or equal to the basic step.

3.2.1 Compute the sum of the first n odd numbers using the formula n^2.

The function you provided correctly computes the sum of the first n odd numbers. This is based on a well-known mathematical fact that the sum of the first n odd numbers is equal to n^2.

1.**Mathematical Insight**: The sum of the first n odd numbers ($1, 3, 5, ..., 2n-1$) can be shown to be equal to n^2. For example:

- Sum of the first 1 odd number: $1=1^2$
- Sum of the first 2 odd numbers: $1+3=4=2^2$
- Sum of the first 3 odd numbers: $1+3+5=9=3^2$ and so forth.

2. **Implementation**: Your implementation correctly reflects this mathematical property. The `if n == 0` check handles the case where no numbers are summed, returning 0 as expected.

```
def sum_of_first_n_odds(n):
    if n == 0:
        return 0
    else:
        return n * n

# Example usage:
n = 8
result = sum_of_first_n_odds(n)
print(f"The sum of the first {n} odd numbers is: {result}")
```

OUTPUT:

```
The sum of the first 5 odd numbers is: 64
```

3.2.2 Compute the sum of the first n even numbers using the formula n*(n+1).

```
def sum_of_first_n_even(n):
    if n == 0:
        return 0
    else:
        return n*(n+1)

# Example usage:
n = 3
result = sum_of_first_n_even(n)
print(f"The sum of the first {n} even numbers is: {result}")
```

OUTPUT:

```
The sum of the first 3 even numbers is: 12
```

3.2.3 Compute the sum of the squares of the first n odd integers

```
def main():
    try:
        # Input from the user
        n = int(input("Enter the number of odd integers: "))

        if n <= 0:
            print("Please enter a positive integer.")
            return
        # Compute the sum of squares
        total_sum = sum((2 * i - 1) ** 2 for i in range(1, n + 1))

        # Output result
        print(f"The sum of the squares of the first {n} odd integers is: {total_sum}")

    except ValueError:
        print("Invalid input. Please enter a valid integer.")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
 Enter the number of odd integers: 3
The sum of the squares of the first 3 odd integers is: 35
```

3.2.4 Compute the sum of the squares of the first n odd integers

```
def sum_of_first_n_natural_numbers(n):
    return n * (n + 1) // 2

# Example usage
n = int(input("Enter the number of natural numbers: "))
print(f"The sum of the first {n} natural numbers is: {sum_of_first_n_natural_numbers(n)}")
```

OUTPUT:

```
Enter the number of natural numbers: 6
The sum of the first 6 natural numbers is: 21
```

3.2.5 Verify the sum of the first n natural numbers using both the basic step and an inductive step.

```
def sum_of_natural_numbers(n):
    """ Compute the sum of the first n natural numbers using the formula."""
    return n * (n + 1) // 2

def verify_inductive_step(k):
    """
    Verify the inductive step for the formula:
    S_n = n * (n + 1) // 2

    Check if the formula holds for k + 1 given it holds for k.
    """
    # Sum for k
    sum_k = sum_of_natural_numbers(k)

    # Sum for k + 1
    sum_k_plus_1 = sum_of_natural_numbers(k + 1)
    # Calculate the sum using the inductive step
    expected_sum_k_plus_1 = sum_k + (k + 1)

    return sum_k_plus_1 == expected_sum_k_plus_1

def main():
    # Basic step
    n = 1
    print(f"Basic step: n = {n}")
    print(f"Sum of the first {n} natural numbers: {sum_of_natural_numbers(n)}")

    # Check the inductive step for first few values
    print("\nVerifying inductive step for k = 1 to 5:")
    for k in range(1, 6):
        result = verify_inductive_step(k)
        print(f"For k = {k}, the formula holds: {result}")

    # General case check
    print("\nGeneral case verification (k + 1) for some k:")
    k = 5  # You can test for other values of k
    if verify_inductive_step(k):
        print(f"The formula holds for k = {k} and k + 1 = {k + 1}.")
    else:
        print(f"The formula does not hold for k = {k} and k + 1 = {k + 1}.")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Basic step: n = 1
Sum of the first 1 natural numbers: 1

Verifying inductive step for k = 1 to 5:
For k = 1, the formula holds: True
For k = 2, the formula holds: True
For k = 3, the formula holds: True
For k = 4, the formula holds: True
For k = 5, the formula holds: True

General case verification (k + 1) for some k:
The formula holds for k = 5 and k + 1 = 6.
```

3.3 Exercise:

1. Compute the sum of the squares of the first n even integers and display the result.

2. Write a program for the Fibonacci sequence can be defined recursively, and also verify that the relationship F(n+2)=F(n+1)+F(n) holds using mathematical induction.

Viva Questions and Answers

1. **What is mathematical induction?**

Answer: Mathematical induction is a method of proof used in mathematics to prove that a statement is true for all natural numbers. It involves two steps: the basic step, where the statement is shown to be true for the first value (usually n=1), and the induction step, where we assume the statement is true for some integer k, and then prove it for k+1.

2. What is the basic step in your proof?

Answer: The base case in this proof is for n=1, where we show that the sum of the first odd number, which is 1, is equal to 1^2.

3. What is the induction hypothesis in this proof?

Answer: The induction hypothesis is the assumption that the formula holds true for some arbitrary integer k. That is, $S(k)=k^2$ for the sum of the first k odd numbers.

4. What is the goal of the induction step?

Answer: The goal of the induction step is to prove that if the formula holds for n=k, it also holds for n=k+1. This is done by showing that $S(k+1)=(k+1)^2$ using the induction hypothesis.

5. Why is mathematical induction a valid proof technique?

Answer: Mathematical induction is valid because it leverages the idea of recursively building on a basic step. By proving that the statement is true for a basic step and that the truth of the statement for n=k implies the truth of the statement for n=k+1, we can conclude that the statement is true for all natural numbers.

6. How does the sum of the first n **odd numbers relate to squares?**

Answer: The sum of the first n odd numbers equals n^2. For example, the sum of the first three odd numbers (1, 3, 5) is 9, which is 3^2.

7. Can you explain the time complexity of calculating permutations and combinations?

Answer: The time complexity of calculating permutations and combinations is primarily dominated by the factorial operations, which grow rapidly. Computing factorials of large numbers can be expensive, but Python's built-in math.factorial function is optimized for performance.

LAB 4: Calculation of Permutation and combination

4.1 Objectives:

Use python

- to calculate the number of permutations and combinations.

4.2 Permutation and combination:

i. Permutations is the arrangement of items where the order matters. The number of permutations of n items taken r at a time is given by:

$$P(n, r) = \frac{n!}{(n-r)!}$$

ii. Combinations refer to the selection of items where the order does not matter. The number of combinations of n items taken r at a time is given by:

$$C(n, r) = \frac{n!}{r! \cdot (n-r)!}$$

4.2.1 Calculate the number of permutations.

```
import math

def permutations(n, r):
    """Calculate the number of permutations of n items taken r at a time."""
    return math.perm(n, r)

# Example usage
n = 6
r = 3
print(f"Permutations of {n} items taken {r} at a time: {permutations(n, r)}")
```

OUTPUT:

```
Permutations of 6 items taken 3 at a time: 120
```

4.2.2 Calculate the number of combinations.

```
import math

def combinations(n, r):
    """Calculate the number of combinations of n items taken r at a time."""
    return math.comb(n, r)

# Example usage
n =6
```

```
r = 3
print(f"Combinations of {n} items taken {r} at a time: {combinations(n, r)}")
```

OUTPUT:

```
Combinations of 6 items taken 3 at a time: 20
```

4.2.3 Write a Python program that calculates and displays both permutations and combinations for a given number of items.

```
import math

def permutations(n, r):
    """Calculate the number of permutations of n items taken r at a time."""
    return math.perm(n, r)

def combinations(n, r):
    """Calculate the number of combinations of n items taken r at a time."""
    return math.comb(n, r)

def main():
    n = int(input("Enter the total number of items (n): "))
    r = int(input("Enter the number of items to choose or arrange (r): "))

    if r > n:
        print("r cannot be greater than n.")
        return

    print(f"Permutations of {n} items taken {r} at a time: {permutations(n, r)}")
    print(f"Combinations of {n} items taken {r} at a time: {combinations(n, r)}")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter the total number of items (n): 5
Enter the number of items to choose or arrange (r): 3
Permutations of 5 items taken 3 at a time: 60
Combinations of 5 items taken 3 at a time: 10
```

4.3 Exercise:

1. Calculate the number of possible passwords of length 4 using a set of 10 unique characters.
2. Determine the number of ways to sample a subset of data from a larger dataset. (Example: A dataset of 50 records and want to choose a sample of 10, calculate the number of possible samples.)
3. Calculate the number of possible genetic variations or gene combinations.
4. Calculate the number of different possible configurations for a puzzle.

Viva Questions and Answers

1. How do you calculate permutations in Python?

Answer: We calculate permutations in Python using the formula $P(n, r)=\frac{n!}{(n-r)!}$. We can use Python's math.factorial function to compute the factorials and then apply the formula.

2. What is the purpose of the math.factorial function in the calculations?

Answer: The math.factorial function computes the factorial of a given number. Factorials are used in permutations and combinations calculations to determine the number of ways to arrange or choose objects.

3. Can you provide an example of calculating permutations for 4 objects taken 2 at a time?

Answer: Using the formula $P(4, 2)=\frac{4!}{(4-2)!}$ =12. In Python, we would call the permutations function as permutations(4, 2), which would return 12.

4. How would you handle cases where r>n in permutations and combinations?

Answer: If r>n, the number of permutations and combinations is zero because you cannot choose or arrange more items than are available. In the permutations and combinations functions, you should check if r>n and handle it appropriately, usually by returning zero or raising an exception.

5. What is the difference between permutations and combinations in terms of order?

Answer: Permutations consider the order of the items, meaning different arrangements of the same items count as distinct permutations. Combinations do not consider order; different arrangements of the same items count as one combination.

6. How do you use Python's itertools module for permutations and combinations?

Answer: Python's itertools module provides functions like itertools.permutations and itertools.combinations for generating permutations and combinations.

7. Can you explain the time complexity of calculating permutations and combinations?

Answer: The time complexity of calculating permutations and combinations is primarily dominated by the factorial operations, which grow rapidly. Computing factorials of large numbers can be expensive, but Python's built-in math.factorial function is optimized for performance.

LAB 5: Implement functions to check whether a given function is one-to-one and onto.

5.1 Objectives:

Use python

- to apply the concept of one-to-one(injective), where a function maps distinct elements of the domain to distinct elements in the codomain.
- to apply the concept of onto(surjective), where a function covers the entire codomain with its outputs

5.2 One-to-one and onto functions:

i. **One-to-One Function (Injective)**:

- A function $f:A\to B$ is called one-to-one (or injective) if different elements in the domain A map to different elements in the codomain B. In other words, if $f(a_1)=f(a_2)$ implies $a_1=a_2$, then f is injective.
- **Mathematically**: $f(a_1)=f(a_2) \implies a_1=a_2$

ii **Onto Function (Surjective)**:

- A function $f:A\to B$ is called onto (or surjective) if every element in the codomain B has a preimage in the domain A. In other words, for every $b\in B$, there is at least one $a\in A$ such that $f(a)=b$.
- **Mathematically**: For every $b\in B$, there exists $a\in A$ such that $f(a)=b$

5.2.1 Check the function $f(x)=x^2$ is one-to-one.

```
def is_injective(domain, function):
    """  Check if the given function is injective (one-to-one).
        :param domain: A set of input values.
    :param function: A function that maps input values to output values.
    :return: True if the function is injective, False otherwise. """
    output_values = set()  # Set to track unique output values
    for x in domain:
        result = function(x)  # Get the function output for input x
        if result in output_values:
            # If result is already in the set, function is not injective
            return False
        output_values.add(result)  # Add result to the set
    return True  # If all outputs are unique, function is injective

# Example function to test
def example_function(x):
    return x * 2  # A simple example function

# Define domain
domain = {1, 2, 3, 4, 5}

# Check injectivity
print("Function is injective:", is_injective(domain, example_function))
```

OUTPUT:

```
Function is injective: True
```

5.2.2 Check the function $f(x)=3x^4 +1$ is onto.

```
def f(x):
    """Function to test: f(x) = 3x^4 + 1."""
    return 3 * x**4 + 1
def is_surjective(codomain):
    """ Check if the function f(x) = 3x^4 + 1 is surjective for a given codomain.
        :param codomain: A set of output values to test.
    :return: True if the function is surjective for the given codomain, False otherwise. """
    # Check if all values in the codomain are greater than or equal to 1
    for y in codomain:
        if y < 1:
            return False
    return True

# Example codomain to test
codomain = {1, 2, 3, 4, 5}  # Example set of values to check
```

```
# Check surjectivity
print("Function f(x) = 3x^4 + 1 is surjective:", is_surjective(codomain))
```

OUTPUT:

```
Function f(x) = 3x^4 + 1 is surjective: True
```

5.2.3 Check the function $g(x) = \begin{cases} x+2 & if\ x \le 0 \\ 2x-1 & if\ x > 0 \end{cases}$ is one-to-one and/or onto.

```
def g(x):
    """Piecewise function to test."""
    if x <= 0:
        return x + 2
    else:
        return 2 * x - 1

def compute_outputs(domain):
    """
    Compute the outputs of the function g(x) for a given domain.

    :param domain: A set of input values.
    :return: A set of output values.
    """
    return {g(x) for x in domain}

def is_injective(domain):
    """
    Check if the function g(x) is injective (one-to-one).

    :param domain: A set of input values.
    :return: True if the function is injective, False otherwise.
    """
    output_values = compute_outputs(domain)
    return len(output_values) == len(domain)

def is_surjective(codomain):
    """
    Check if the function g(x) is surjective for a given codomain.

    :param codomain: A set of output values to test.
    :return: True if the function is surjective for the given codomain, False otherwise.
    """
    # Check surjectivity by solving for x given y in the codomain
    for y in codomain:
        # Solve the equations based on the piecewise function
        # For x <= 0: y = x + 2 -> x = y - 2
        # For x > 0: y = 2x - 1 -> x = (y + 1) / 2

        if y >= -2 and y <= 1:  # Possible range for the first piece
            continue
        if y >= -1:  # Possible range for the second piece
            continue
        return False
    return True

def main():
    # Define domain and codomain
    domain = {-3, -1, 0, 1, 2}  # Example domain values
    codomain = {-1, 0, 1, 2, 3}  # Example codomain values

    # Check injectivity
    injectivity_result = is_injective(domain)
    print("Function g(x) is injective:", injectivity_result)

    # Check surjectivity
    surjectivity_result = is_surjective(codomain)
    print("Function g(x) is surjective:", surjectivity_result)

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Function g(x) is injective: False
Function g(x) is surjective: True
```

5.3 Exercise:

1. Check the function j(x)= $4x-7$ is one-to-one or onto.
2. Verify if t(x)=log(x+1) is injective (one-to-one) over the domain [0,10].
3. Determine if $f(x) = \frac{1}{x+1}$ is surjective (onto) for the codomain (0,∞).

Viva Questions and Answers

1. **What does it mean for a function to be one-to-one (injective)?**
Answer: A function is one-to-one if different elements in the domain map to different elements in the codomain. In other words, if $f(x_1)=f(x_2)$ implies $x_1=x_2$, then the function is injective.

2. How can you determine if a function is one-to-one using a Python program?
Answer: To determine if a function is one-to-one, you can check if any two different elements in the domain map to the same value in the codomain. This can be done by iterating through the domain and checking if any output value repeats.

3. What does it mean for a function to be onto (surjective)?
Answer: A function is onto if every element in the codomain is mapped to by at least one element in the domain. This means that the range of the function is equal to the codomain.

4. How can you determine if a function is onto using a Python program?
Answer: To determine if a function is onto, we can compare the set of values that the function maps to with the codomain. If every element in the codomain is included in the set of mapped values, then the function is surjective.

5. Can you provide an example of a function that is one-to-one but not onto?
Answer: Consider the function $f(x)=x^2$ where the domain is {1,2,3} and the codomain is {1,4,9}. This function is one-to-one but not onto if the codomain is set to {1,4} instead of {1,4,9}, as 9 is not mapped to by any element in the domain.

6. Can you give an example of a function that is onto but not one-to-one?
Answer: Consider the function f(x)=mod(x,2) where the domain is {1,2,3} and the codomain is {0,1}. This function is onto because every element in the codomain {0,1} is covered. However, it is not one-to-one because both 1 and 3 map to the same value 1.

7. How would you modify the program to handle functions with larger domains and codomains?
Answer: The program can handle larger domains and codomains by using efficient data structures like sets and dictionaries. The implementation will scale as long as the domain and codomain are correctly represented and the function is defined over these sets.

8. What is the difference between checking for injectivity and surjectivity?
Answer: Checking for injectivity involves ensuring that no two different elements in the domain map to the same element in the codomain, while checking for surjectivity involves ensuring that every element in the codomain is covered by some element in the domain.

9. Why is it important to understand the concepts of injective and surjective functions?
Answer: Understanding injective and surjective functions is important in various fields such as mathematics, computer science, and engineering. These concepts are crucial for designing algorithms, understanding function behavior, and ensuring that functions meet specific criteria for applications.

10. How can you generalize the concept of injectivity and surjectivity to functions with more complex structures?
Answer: For more complex structures, such as functions defined over multiple variables or functions between different types of sets, the concepts of injectivity and surjectivity still apply. The general principles are the same: injectivity requires unique mapping for each element in the domain, and surjectivity requires complete coverage of the codomain. In higher dimensions or more complex structures, these concepts are extended to ensure the function's properties meet the required criteria.

LAB 6: Check whether the relation is equivalence or not.

6.1 Objectives:

Use python

- to apply the definitions of reflexivity, symmetry, and transitivity.
- to find the equivalence classes for the relation R on the set A.

6.2 Equivalence Relation:

An equivalence relation is a type of relation on a set that satisfies three properties: reflexivity, symmetry, and transitivity.

Properties of an Equivalence Relation

1. **Reflexivity**: Every element is related to itself. For a relation R on a set A, this means for every a ∈A, the pair (a, a) must be in R.
2. **Symmetry**: If an element a is related to b, then b must also be related to a. For (a, b)∈R, the pair (b, a) must also be in R.
3. **Transitivity**: If an element a is related to b, and b is related to c, then a must also be related to c. For (a, b)∈R and (b, c)∈R, the pair (a, c) must also be in R.

6.2.1 Check whether a given relation on a set is an equivalence relation by verifying reflexivity, symmetry, and transitivity. (Example: A = {1, 2, 3} R = {(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)}).

```
def is_reflexive(A, R):
    """Check if the relation R on set A is reflexive."""
    for a in A:
        if (a, a) not in R:
            return False
    return True

def is_symmetric(R):
    """Check if the relation R is symmetric."""
    for (a, b) in R:
        if (b, a) not in R:
            return False
    return True

def is_transitive(A, R):
    """Check if the relation R on set A is transitive."""
    for (a, b) in R:
        for (c, d) in R:
            if b == c and (a, d) not in R:
                return False
    return True

def check_equivalence_relation(A, R):
    """Check if the relation R on set A is an equivalence relation."""
    if is_reflexive(A, R) and is_symmetric(R) and is_transitive(A, R):
        return True
    return False

# Example usage
A = {1, 2, 3}
R = {(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)}

if check_equivalence_relation(A, R):
    print("The relation R is an equivalence relation.")
else:
    print("The relation R is not an equivalence relation.")
```

OUTPUT:

```
The relation R is an equivalence relation.
```

6.2.2 Find the equivalence classes for relation R on set A. Given that A = {1, 2, 3} R = {(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)}.

```
def find_equivalence_classes(A, R):
    """Find the equivalence classes for relation R on set A."""
    classes = [ ]
    visited = set( )

    for a in A:
        if a not in visited:
            # Find all elements related to a
            class_members = {b for (x, b) in R if x == a}
            class_members.update({x for (x, b) in R if b == a})
            classes.append(class_members)
            visited.update(class_members)

    return classes

# Example usage
A = {1, 2, 3}
R = {(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)}

equivalence_classes = find_equivalence_classes(A, R)
print("Equivalence classes:", equivalence_classes)
```

OUTPUT:

```
Equivalence classes: [{1, 2}, {3}]
```

6.2.3 Generate the relation R from equivalence classes. Given that classes = [{1, 2}, {3}].

```
def generate_relation_from_classes(classes):
    """Generate the relation R from equivalence classes."""
    R = set()
    for cls in classes:
        for a in cls:
            for b in cls:
                R.add((a, b))
    return R

# Example usage
classes = [{1, 2}, {3}]
R = generate_relation_from_classes(classes)
print("Generated relation R:", R)
```

OUTPUT:

```
Generated relation R: {(1, 2), (2, 1), (1, 1), (3, 3), (2, 2)}
```

6.2.4 Check if two people are equivalent if they have the same name and age.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

def are_equivalent(person1, person2):
    return person1.name == person2.name and person1.age == person2.age

p1 = Person("RAMA", 45)
p2 = Person("RAMA", 45)
p3 = Person("LAKSHMANA", 30)

print(are_equivalent(p1, p2))  # True
print(are_equivalent(p1, p3))  # False
```

OUTPUT:

```
True
False
```

6.3 Exercise:

1. Given the set A={1,2,3} and the relation R={(1,1),(2,2),(1,2)}, check whether the given relation R on set A is an equivalence relation or not.
2. Write a program on a group emails that are similar (or equivalent) to detect spam.

Viva Questions and Answers

1. **What is an equivalence relation?**

Answer: An equivalence relation on a set is a relation that satisfies three properties: reflexivity, symmetry, and transitivity. These properties ensure that the relation partitions the set into equivalence classes.

2. How can you check if a relation is reflexive?

Answer: A relation is reflexive if every element in the set is related to itself. In terms of ordered pairs, for every element x in the set, the pair (x,x) should be in the relation.

3. How can you check if a relation is symmetric?

Answer: A relation is symmetric if for every pair (a,b) in the relation, the pair (b,a) must also be in the relation. This means that if a is related to b, then b must be related to a.

4. How can you check if a relation is transitive?

Answer: A relation is transitive if whenever (a,b) and (b,c) are in the relation, the pair (a,c) must also be in the relation. This ensures that the relation can be "chained" through intermediate elements.

5. Can you provide an example of an equivalence relation?

Answer: An example of an equivalence relation is the relation of equality on any set. For any set A the relation $\{(a,a)|a\in A\}$ is reflexive, symmetric, and transitive, making it an equivalence relation.

6. What are equivalence classes?

Answer: Equivalence classes are subsets of a set where each subset contains elements that are equivalent to each other under a given equivalence relation. For example, if a is equivalent to b and b is equivalent to c, then a, b, and c belong to the same equivalence class.

7. Why is it important to check if a relation is an equivalence relation?

Answer: Checking if a relation is an equivalence relation is important because it helps in understanding how elements are grouped or classified within a set. Equivalence relations are fundamental in various fields such as mathematics, computer science, and logic, where they are used to define partitions, classes, and structures.

8. How would you modify the Python program to handle relations on larger sets or more complex structures?

Answer: To handle larger sets or more complex structures, ensure that the relation and the set of elements are represented efficiently. The program should be able to manage larger data structures, and you might need to optimize the implementation for performance if working with very large sets.

9. Can you explain the significance of each property (reflexive, symmetric, transitive) in the context of equivalence relations?

Answer:

- **Reflexive**: Ensures every element is related to itself, establishing a base point for comparison.
- **Symmetric**: Ensures that the relationship is mutual, reflecting a bidirectional connection.
- **Transitive**: Ensures that the relationship is consistent and can be extended through intermediates, providing a coherent structure for equivalence classes.

10. How does the concept of equivalence relations apply in computer science?

Answer: In computer science, equivalence relations are used in various areas such as database design (for defining normalization), algorithms (for partitioning and clustering), and formal languages (for defining equivalence classes of strings). They help in structuring data and simplifying complex problems.

LAB 7: Implement the Fibonacci sequence using both an iterative approach and a recursive approach.

7.1 Objectives:

Use python

- to calculate the Fibonacci sequence both iterative and recursive approaches.

7.2 Fibonacci sequence:

It is a series of numbers where each number is the sum of the two preceding ones. The sequence starts with 0 and 1, and the subsequent numbers are generated by adding the two preceding numbers.

Fibonacci Sequence Definition

The Fibonacci sequence F(n) is defined as: $F(0)=0$, $F(1)=1$, $F(n)=F(n-1)+F(n-2)$ for $n\geq 2$.

Generating the Fibonacci Sequence

There are two common approaches to generate the Fibonacci sequence:

1. **Iterative Approach**:

   Uses a loop to calculate each Fibonacci number up to n.

2. **Recursive Approach**:

   Uses a recursive function to calculate Fibonacci numbers.

7.2.1 Implement the Fibonacci sequence in Python using iterative approach.

```
def fibonacci_iterative_with_steps(n):
    """Compute the nth Fibonacci number using an iterative approach and show intermediate steps."""
    if n <= 0:
        return "Invalid input, n should be a positive integer."
    elif n == 1:
        return 0
    elif n == 2:
        return 1

    a, b = 0, 1
    print(f"Iteration 1: {a}")
    print(f"Iteration 2: {b}")
    for i in range(2, n):
        a, b = b, a + b
        print(f"Iteration {i + 1}: {b}")
    return b

# Example usage:
n = 10
print(f"Fibonacci number at position {n} (iterative): {fibonacci_iterative_with_steps(n)}")
```

OUTPUT:

```
Iteration 1: 0
Iteration 2: 1
Iteration 3: 1
Iteration 4: 2
Iteration 5: 3
Iteration 6: 5
Iteration 7: 8
Iteration 8: 13
Iteration 9: 21
Iteration 10: 34
Fibonacci number at position 10 (iterative): 34
```

7.2.2 Implement the Fibonacci sequence in Python using recursive approach.

```
def fibonacci_recursive(n):
    """Compute the nth Fibonacci number using a recursive approach."""
    if n <= 0:
        return "Invalid input, n should be a positive integer."
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)

# Example usage:
n = 8
print(f"Fibonacci number at position {n} (recursive): {fibonacci_recursive(n)}")
```

OUTPUT:

```
Fibonacci number at position 8 (recursive): 13
```

7.3 Exercise:

1. Write a Python program that calculates the sum of the first n Fibonacci numbers using both iterative and recursive approaches. Compare its efficiency with the iterative approach.

Viva Questions and Answers

1. What is the Fibonacci sequence?

Answer: The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, starting from 0 and 1. The sequence typically starts as 0, 1, 1, 2, 3, 5, 8, 13, etc.

2. How does the iterative approach to generating the Fibonacci sequence work?

Answer: The iterative approach uses a loop to compute the Fibonacci numbers sequentially. It starts with the first two numbers and repeatedly calculates the next number by summing the last two numbers in the sequence until the desired length is reached.

3. How does the recursive approach to generating the Fibonacci sequence work?

Answer: The recursive approach calculates the Fibonacci sequence by defining the sequence in terms of itself. It calls the function recursively to find the previous two numbers and then sums them to get the next number, building the sequence from the base cases.

4. What are the advantages of the iterative approach over the recursive approach for generating Fibonacci numbers?

Answer: The iterative approach is generally more efficient in terms of both time and space because it avoids the overhead of multiple function calls and stack space used in recursion. It is less likely to run into issues with stack overflow for large inputs.

5. What are the disadvantages of the recursive approach to generating Fibonacci numbers?

Answer: The recursive approach can be less efficient due to repeated calculations and the overhead of function calls. For large n, it can lead to exponential time complexity and stack overflow errors because of deep recursion.

6. How can you optimize the recursive approach to improve performance?

Answer: The recursive approach can be optimized using memorization or dynamic programming. By storing previously computed values and reusing them, you can avoid redundant calculations and significantly improve performance.

7. Can you provide an example of how the iterative approach is implemented in Python?

Answer: Yes, the iterative approach initializes the first two numbers and uses a loop to compute subsequent numbers by summing the last two numbers, adding each result to the sequence until the desired length is reached.

8. Can you provide an example of how the recursive approach is implemented in Python?

Answer: Yes, the recursive approach defines the base cases for $n \le 2$ and calls itself to compute the sequence up to $n-1$, appending the sum of the last two numbers to build the sequence.

9. What are some real-world applications of the Fibonacci sequence?

Answer: The Fibonacci sequence appears in various natural phenomena and applications, such as the branching patterns of trees, the arrangement of leaves on a stem, the reproduction patterns of rabbits, and in algorithms related to computer science, such as sorting and searching.

10. How would you modify the Python programs to handle very large inputs efficiently?

Answer: For very large inputs, you would use the iterative approach or an optimized recursive approach with memorization. For extremely large numbers, consider using specialized libraries or algorithms designed to handle large integers efficiently and avoid stack overflow issues.

LAB 8: Write a program to verify a given relation forms a partial order or not.

8.1 Objectives:

Use python

- to verify if the given relation is a partial order.
- to generate and plot the Hasse diagram of the partial order.

8.2 Partial Order:

To determine whether a given relation on a set forms a partial order, it must satisfy three properties:

1. Reflexive: For every element a ∈A, the relation (a, a) must be in R.
2. Anti-symmetry: For every pair (a, b) ∈R and (b, a) ∈R, a must be equal to b.
3. Transitivity: For every pair (a, b) ∈R and (b, c) ∈R(b, c) , the pair (a, c) must also be in R.

8.2.1 Check whether a given relation on a set satisfies the conditions of a partial order.

```
class PartialOrder:
    def __init__(self, elements, relation):
        """
        Initialize a partial order.

        :param elements: A list of elements in the set.
        :param relation: A set of tuples representing the partial order relation.
        """
        self.elements = elements
        self.relation = relation

    def is_reflexive(self):
        """
        Check if the relation is reflexive.
        """
        for element in self.elements:
            if (element, element) not in self.relation:
                return False
        return True

    def is_antisymmetric(self):
        """
        Check if the relation is antisymmetric.
        """
        for (a, b) in self.relation:
            if (b, a) in self.relation and a != b:
                return False
        return True

    def is_transitive(self):
        """
        Check if the relation is transitive.
        """
        for (a, b) in self.relation:
            for (c, d) in self.relation:
                if b == c and (a, d) not in self.relation:
                    return False
        return True

    def check_partial_order(self):
        """
        Check if the relation satisfies all properties of a partial order.
        """
        return (self.is_reflexive() and
                self.is_antisymmetric() and
                self.is_transitive())

# Example usage
elements = [1, 2, 3]
relation = {(1, 1), (2, 2), (3, 3), (1, 2), (1, 3), (2, 3)}

po = PartialOrder(elements, relation)

print("Elements:", po.elements)
print("Relation:", po.relation)
print("Is reflexive:", po.is_reflexive())
print("Is antisymmetric:", po.is_antisymmetric())
print("Is transitive:", po.is_transitive())
print("Is a partial order:", po.check_partial_order())
```

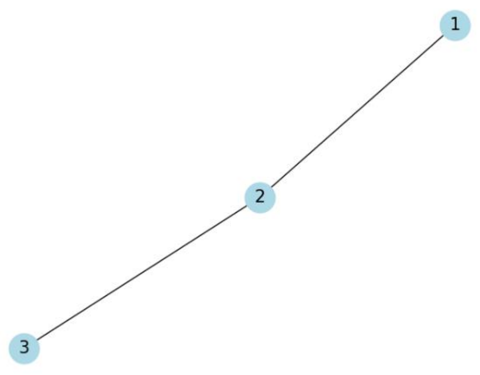
OUTPUT:

```
Elements: [1, 2, 3]
Relation: {(2, 3), (1, 2), (3, 3), (2, 2), (1, 1), (1, 3)}
Is reflexive: True
Is antisymmetric: True
Is transitive: True
Is a partial order: True
```

8.2.2 Verify Partial Order and draw Hasse diagram for the given relation = [(1, 1), (1, 2), (2, 2), (2, 3), (3, 3)].

```
import networkx as nx
import matplotlib.pyplot as plt

class PartialOrder:
    def __init__(self, elements, relation):
        self.elements = elements
        self.relation = relation

    def is_reflexive(self):
        return all((elem, elem) in self.relation for elem in self.elements)

    def is_antisymmetric(self):
        return all((b, a) not in self.relation or a == b for (a, b) in self.relation)

    def is_transitive(self):
        return all((a, d) in self.relation for (a, b) in self.relation for (c, d) in self.relation
     if b == c)

    def draw_hasse_diagram(self):
        G = nx.DiGraph()
        G.add_nodes_from(self.elements)
        for (a, b) in self.relation:
            if a != b:
                G.add_edge(a, b)

        pos = nx.spring_layout(G, seed=42)
        nx.draw(G, pos, with_labels=True, arrows=False, node_size=700,
     node_color='lightblue', font_size=15)
        plt.title("Hasse Diagram")
        plt.show()

# Example usage:
elements = [1, 2, 3]
relation = [(1, 1), (1, 2), (2, 2), (2, 3), (3, 3)]

po = PartialOrder(elements, relation)
po.draw_hasse_diagram()
```

OUTPUT:



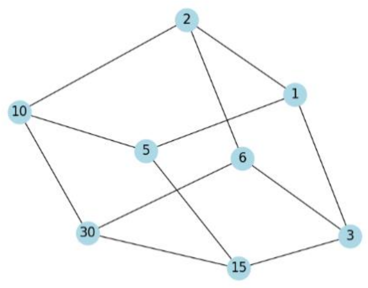
8.2.3 Generate and visualize the Hasse diagram to represent the partial order of the divisor 30.

```
import networkx as nx
import matplotlib.pyplot as plt

def find_divisors(n):
    """
    Function to find all positive divisors of a given number n.
    """
    divisors = [ ]
    for i in range(1, n + 1):
        if n % i == 0:
            divisors.append(i)
    return divisors

def create_partial_order(divisors):
    """
    Create a list of pairs representing the partial order relation.
    Each pair (a, b) indicates that a divides b.
    """
    relation = [ ]
    for i in range(len(divisors)):
        for j in range(i + 1, len(divisors)):
            if divisors[j] % divisors[i] == 0:
                relation.append((divisors[i], divisors[j]))
    return relation

def draw_hasse_diagram(elements, relation):
    """
    Draw the Hasse diagram of a partial order.
    """
    G = nx.DiGraph( )
    G.add_nodes_from(elements)

    # Add edges while avoiding transitive edges
    for (a, b) in relation:
        if a != b:
            # Check if (a, b) is a transitive edge
            if not any((a, c) in relation and (c, b) in relation for c in elements if c != a and c != b):
                G.add_edge(a, b)

    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=True, arrows=False, node_size=700, node_color='lightblue',
font_size=15)
    plt.title("Hasse Diagram of Positive Divisors")
    plt.show( )

# Main program
number = 30
divisors = find_divisors(number)
```

```
relation = create_partial_order(divisors)

print(f"The positive divisors of {number} are: {divisors}")
print(f"The relation (a divides b) is: {relation}")

# Draw the Hasse diagram
draw_hasse_diagram(divisors, relation)
```

OUTPUT:

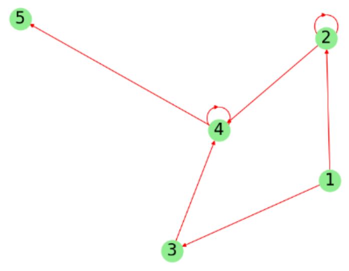
```
The positive divisors of 30 are: [1, 2, 3, 5, 6, 10, 15, 30]
The relation (a divides b) is: [(1, 2), (1, 3), (1, 5), (1, 6), (1, 10), (1, 15), (1, 30), (2, 6), (2, 10), (2, 30),
(3, 6), (3, 15), (3, 30), (5, 10), (5, 15), (5, 30), (6, 30), (10, 30), (15, 30)]
```



8.2.4 Create a directed graph (DiGraph) with nodes and directed edges.

```
import networkx as nx
import matplotlib.pyplot as plt

def create_directed_graph():
    """
    Create a directed graph (DiGraph) with nodes and directed edges.
    """
    # Create a directed graph
    G = nx.DiGraph( )

    # Add nodes
    G.add_nodes_from([1, 2, 3, 4, 5])

    # Add directed edges
    G.add_edges_from([(1, 2),(2,2), (1, 3), (2, 4), (3, 4), (4,4), (4, 5)])

    return G

def draw_directed_graph(G):
    """
    Draw the directed graph using matplotlib.
    """
    pos = nx.spring_layout(G, seed=4)  # positions for all nodes

    # Draw the graph
    nx.draw(G, pos, with_labels=True, arrows=True, node_size=700, node_color='lightgreen',
font_size=20, edge_color='red')

    # Add title
    plt.title("Directed Graph (DiGraph) Visualization")

    # Show plot
    plt.show( )

# Main program
if __name__ == "__main__":
    # Create the directed graph
    G = create_directed_graph( )

    # Draw the directed graph
    draw_directed_graph(G)
```

OUTPUT:



8.3 Exercise:

1. Generate and visualize the Hasse diagram to represent the partial order of the divisor 42.
2. Let A={1,2,3,4,6} and R be a relation on A define by aRb if and only if "a is multiple of b" represent the relation R as a matrix , draw its digraph and relation R.

Viva Questions and Answers

1. What is a partial order?

Answer: A partial order is a relation on a set that is reflexive, antisymmetric, and transitive. It generalizes the concept of ordering and allows some elements to be incomparable.

2. How can you verify if a relation is reflexive?

Answer: A relation is reflexive if every element in the set is related to itself. In terms of ordered pairs, for every element x in the set, the pair (x, x) should be in the relation.

3. What is the antisymmetric property in the context of partial orders?

Answer: The antisymmetric property states that if (a, b) and (b, a) are both in the relation, then a must be equal to b. This prevents elements from being mutually related unless they are identical.

4. How can you verify if a relation is antisymmetric?

Answer: A relation is antisymmetric if, for every pair (a, b) in the relation, if (b, a) is also in the relation, then a must be equal to b.

5. What is the transitive property in the context of partial orders?

Answer: The transitive property states that if (a, b) and (b, c) are in the relation, then (a, c) must also be in the relation. This ensures that the relation is consistent and can be extended through intermediate elements.

6. How can you verify if a relation is transitive?

Answer: A relation is transitive if, for every pair (a, b) and (b, c) in the relation, the pair (a, c) must also be in the relation.

7. Can you provide an example of a relation that forms a partial order?

Answer: An example of a partial order is the "less than or equal to" relation (≤) on the set of real numbers. It is reflexive (every number is less than or equal to itself), antisymmetric (if $a \leq b$ and $b \leq a$, then $a=b$), and transitive (if $a \leq b$ and $b \leq c$, then $a \leq c$).

8. How would you modify the Python program to handle larger sets or relations?

Answer: To handle larger sets or relations, ensure that the relation and the set of elements are represented efficiently. The program should be capable of processing larger data structures and may need optimization for performance if dealing with very large sets.

9. What is the significance of checking if a relation is a partial order?

Answer: Checking if a relation is a partial order is important for understanding the structure and properties of the set. It is used in various fields such as mathematics, computer science, and database theory to define and analyze hierarchical structures and ordering systems.

10. How does the concept of partial order apply in computer science?

Answer: In computer science, partial orders are used in scheduling problems, data dependency analysis, hierarchical data structures (like trees and graphs), and in database design to manage and query relational data.

LAB 9: Write a program on assign colors to the vertices of a graph, no two adjacent vertices share the same color.

9.1 Objectives:

Use python

- to implement the Greedy Coloring Algorithm to assign colors to the vertices of a graph such that no two adjacent vertices share the same color.

9.2 Greedy Coloring Algorithm:

The **Greedy Coloring Algorithm** is a straightforward approach to graph coloring that assigns colors to vertices of a graph such that no two adjacent vertices have the same color. Here's a step-by-step description of the algorithm:

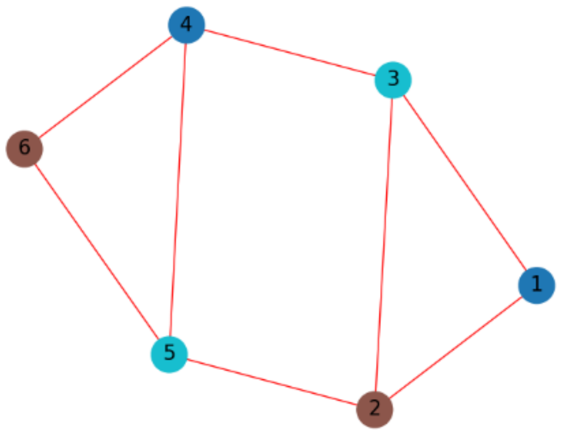
1. **Initialize**:
   - ★ **Create an empty list or dictionary** to store the color assigned to each vertex.
   - ★ **Initialize** an array to keep track of the available colors.
2. **Order the Vertices**:
   - ★ For simplicity, we can process the vertices in any arbitrary order. However, in practice, ordering vertices by degree or other heuristics can improve performance.
3. **Color Assignment**:
   - ★ For each vertex v in the graph:
     1. **Determine the Colors of Adjacent Vertices**:
        - ▪ Check the colors of all vertices adjacent to v.
     2. **Assign the Smallest Available Color**:
        - ▪ Find the smallest color number that is not used by the adjacent vertices.
        - ▪ Assign this color to vertex v.
4. **Output the Coloring**:
   - ★ Return or print the color assignments for all vertices.

9.2.1 Greedy algorithm to color the graph G such that no two adjacent vertices share the same color.

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

def greedy_coloring(G):
    """
    Greedy algorithm to color the graph G such that no two adjacent vertices share the same color.
    """
    colors = {}
    for node in G.nodes():
        # Find the colors of adjacent nodes
        adjacent_colors = set(colors.get(neighbor) for neighbor in G.neighbors(node))
        # Assign the lowest color that is not used by adjacent nodes
        color = 1
        while color in adjacent_colors:
            color += 1
        colors[node] = color
    return colors

def draw_graph_with_coloring(G, colors):
    """
    Draw the graph with vertices colored according to the colors dictionary.
    """
    pos = nx.spring_layout(G, seed=30)  # positions for all nodes
    color_map = [colors.get(node) for node in G.nodes()]

    nx.draw(G, pos, with_labels=True, node_color=color_map, node_size=700,
cmap=plt.get_cmap('tab10'), font_size=15, edge_color='red')
    plt.title("Graph Coloring with Greedy Algorithm")
    plt.show( )

def main( ):
    # Create a sample graph
    G = nx.Graph( )
    G.add_edges_from([(1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 6), (4, 6), (2, 5)])

    # Apply the greedy coloring algorithm
    colors = greedy_coloring(G)

    # Print the color assigned to each vertex
    print("Vertex colors:")
    for node, color in colors.items( ):
        print(f"Vertex {node}: Color {color}")

    # Draw the graph with colors
    draw_graph_with_coloring(G, colors)

if __name__ == "__main__":
    main( )
```

OUTPUT:

```
Vertex colors:
Vertex 1: Color 1
Vertex 2: Color 2
Vertex 3: Color 3
Vertex 4: Color 1
Vertex 5: Color 3
Vertex 6: Color 2
```



9.2.2 Greedy Algorithm for Activity Selection

```
def activity_selection(activities):
    """
    Select the maximum number of non-overlapping activities.
    :param activities: List of tuples (start_time, end_time)
    :return: List of selected activities
    """
    # Sort activities based on their finish times
    activities.sort(key=lambda x: x[1])

    selected_activities = []
    last_end_time = 0

    for activity in activities:
        start, end = activity
        if start >= last_end_time:
            selected_activities.append(activity)
            last_end_time = end

    return selected_activities

def main():
    # Example activities (start_time, end_time)
    activities = [(1, 4), (2, 6), (3, 5), (5, 7), (8, 9), (5, 9)]
    selected = activity_selection(activities)

    print("Selected activities:")
    for activity in selected:
        print(activity)

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Selected activities:
(1, 4)
(5, 7)
(8, 9)
```

9.3 Exercise:

1. Find the minimum number of coins required to make the given amount using Greedy algorithm.

Viva Questions and Answers

1. What is graph coloring?

Answer: Graph coloring is the assignment of colors to the vertices of a graph such that no two adjacent vertices share the same color. The goal is to minimize the number of colors used while satisfying this constraint.

2. What is the greedy algorithm for graph coloring?

Answer: The greedy algorithm for graph coloring assigns colors to vertices one by one, choosing the smallest possible color that is not used by adjacent vertices. This approach does not guarantee the optimal number of colors but provides a valid coloring.

3. How does the `is_safe` function work in the graph coloring program?

Answer: The `is_safe` function checks if it's safe to assign a particular color to a vertex by ensuring that no adjacent vertices have the same color. It returns `True` if the color can be assigned safely, and `False` otherwise.

4. What is the purpose of the `graph_coloring_util` function in the program?

Answer: The `graph_coloring_util` function is a recursive utility function used to assign colors to vertices. It tries to color the current vertex and recursively proceeds to the next vertex. If it encounters a conflict, it backtracks and tries a different color.

5. What does the `graph_coloring` function do?

Answer: The `graph_coloring` function initializes the coloring process, sets up the color array, and calls the recursive utility function to try and color the graph with the given number of colors. It returns whether a valid coloring was found and the color assignment if successful.

6. How can you determine if a graph can be colored with a given number of colors?

Answer: You can determine if a graph can be colored with a given number of colors by attempting to assign colors using an algorithm, such as the greedy algorithm or backtracking. If a valid coloring is found, the graph can be colored with that number of colors.

7. What is the significance of the adjacency matrix in the graph coloring problem?

Answer: The adjacency matrix is a representation of the graph where the element at row iii and column j indicates whether there is an edge between vertex I and vertex j. It is used to check adjacency when assigning colors.

8. Can the greedy algorithm always find the optimal number of colors for a graph?

Answer: No, the greedy algorithm does not always find the optimal number of colors. It provides a valid coloring but may use more colors than necessary. Finding the optimal number of colors is a more complex problem, often requiring advanced techniques.

9. What is the time complexity of the greedy graph coloring algorithm?

Answer: The time complexity of the greedy graph coloring algorithm depends on the graph representation and implementation. For the provided algorithm, it can be $O(V^2)$ where V is the number of vertices, due to the need to check each vertex and its adjacent vertices.

10. How would you modify the program to handle larger graphs or different types of input?

Answer: To handle larger graphs or different input types, you would need to optimize the algorithm for performance, potentially using more efficient data structures or algorithms. We could also implement more advanced coloring algorithms or heuristics depending on the problem's requirements.

LAB 10: Implement the Traveling Salesman Problem (TSP) using a Hamilton path approach to find the shortest Hamilton Path in a weighted graph.

10.1 Objectives:

Use python

- to determine the shortest Hamiltonian path in a weighted graph using a brute-force approach.

10.2 Brute-Force Method:

This method involves generating all permutations of vertices, calculating the path cost for each permutation, and selecting the one with the minimum cost. This method is feasible for small graphs due to its factorial time complexity.

10.2.1 Find the shortest Hamiltonian path in a weighted graph using a brute-force approach

```
import itertools

def shortest_hamiltonian_path(weights):
    """
    Find the shortest Hamiltonian path in a weighted graph using a brute-force approach.

    :param weights: A 2D list (adjacency matrix) where weights[i][j] represents the cost to travel from
node i to node j.
    :return: The minimum cost of the Hamiltonian path and the corresponding path.
    """
    n = len(weights)
    min_cost = float('inf')
    best_path = [ ]

    # Generate all permutations of nodes
    for perm in itertools.permutations(range(n)):
        # Calculate the cost of the current permutation path
        current_cost = 0
        valid_path = True
        for i in range(n - 1):
            if weights[perm[i]][perm[i + 1]] == float('inf'):
                valid_path = False
                break
            current_cost += weights[perm[i]][perm[i + 1]]

        # Check if this path is valid and update the minimum cost
        if valid_path and current_cost < min_cost:
            min_cost = current_cost
            best_path = perm

    return min_cost, best_path

# Example usage:
weights = [
    [0, 5, 15, 20],
    [5, 0, 35, 25],
    [15, 35, 0, 10],
    [20, 25, 10, 0]
]

min_cost, best_path = shortest_hamiltonian_path(weights)
print(f"Minimum cost of Hamiltonian path: {min_cost}")
print(f"Best path: {best_path}")
```

OUTPUT:

```
Minimum cost of Hamiltonian path: 30
Best path: (1, 0, 2, 3)
```

10.2.2 Write a program to optimize the placement of cell towers to cover all geographic regions with minimal total installation and maintenance costs using TSP.

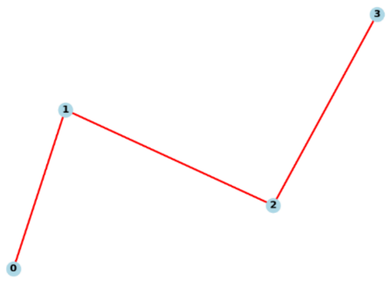
```
import itertools
import math
import matplotlib.pyplot as plt
import networkx as nx

def euclidean_distance(loc1, loc2):
    return math.sqrt((loc1[0] - loc2[0]) ** 2 + (loc1[1] - loc2[1]) ** 2)

def create_distance_matrix(locations):
    n = len(locations)
    distance_matrix = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(i + 1, n):
            distance = euclidean_distance(locations[i], locations[j])
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance
    return distance_matrix

def calculate_path_cost(path, distance_matrix):
    cost = 0
    for i in range(len(path) - 1):
        cost += distance_matrix[path[i]][path[i + 1]]
    return cost
```

```
def find_shortest_tsp_path(distance_matrix):
    n = len(distance_matrix)
    min_cost = float('inf')
    best_path = None

    for perm in itertools.permutations(range(n)):
        current_cost = calculate_path_cost(perm, distance_matrix)
        if current_cost < min_cost:
            min_cost = current_cost
            best_path = perm

    return min_cost, best_path

def draw_tower_placement(locations, path):
    G = nx.Graph()
    n = len(locations)

    # Add nodes and edges
    for i in range(n):
        G.add_node(i, pos=locations[i])
    for i in range(n):
        for j in range(i + 1, n):
            if (i, j) in path or (j, i) in path:
                G.add_edge(i, j, weight=euclidean_distance(locations[i], locations[j]))

    pos = {i: locations[i] for i in range(n)}
    edge_labels = nx.get_edge_attributes(G, 'weight')

    plt.figure(figsize=(10, 7))
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=500, font_size=16,
              font_weight='bold', edge_color='gray', linewidths=2, width=2)
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

    if path:
        path_edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='red', width=3)

    plt.title('Optimal Cell Tower Placement')
    plt.show()

# Example locations (latitude, longitude)
locations = [
    (0, 0),
    (1, 5),
    (5, 2),
    (7, 8)
]

distance_matrix = create_distance_matrix(locations)
min_cost, best_path = find_shortest_tsp_path(distance_matrix)
```

```
print(f"Minimum cost of cell tower placement: {min_cost}")
print(f"Optimal path: {best_path}")

draw_tower_placement(locations, best_path)
```

OUTPUT:

```
Minimum cost of cell tower placement: 16.423574833929543
Optimal path: (0, 1, 2, 3)
```



10.3 Exercise:

1. Develop a Python program to find the shortest route that a fleet of trucks should take to deliver goods to multiple locations, minimizing the total distance traveled.

Viva Questions and Answers

1. What is the Traveling Salesman Problem (TSP)?

Answer: The Traveling Salesman Problem (TSP) is a problem in which a salesman must find the shortest possible route that visits each city exactly once and returns to the origin city. It is a classic optimization problem in computer science and operations research.

2. What is a Hamiltonian path?

Answer: A Hamiltonian path is a path in a graph that visits each vertex exactly once. In the context of TSP, finding a Hamiltonian path means finding a route that visits each city exactly once.

3. How does the brute-force approach work for solving TSP using a Hamiltonian path?

Answer: The brute-force approach generates all possible permutations of the vertices (cities) and calculates the path length for each permutation. It then selects the permutation with the minimum path length as the solution.

4. What is the time complexity of the brute-force TSP solution?

Answer: The time complexity of the brute-force solution is O(n!), where n is the number of vertices. This is because there are n! possible permutations of the vertices, and each permutation requires computing the path length.

5. What is the `calculate_path_length` function used for in the program?

Answer: The `calculate_path_length` function calculates the total length of a given path by summing the weights of the edges between consecutive vertices in the path.

6. What are the limitations of using the brute-force approach for TSP?

Answer: The brute-force approach is impractical for large graphs because the number of permutations grows factorially with the number of vertices. It is only feasible for small instances due to its exponential time complexity.

7. How can the brute-force approach be improved or optimized?

Answer: For larger instances of TSP, more efficient algorithms and heuristics are used, such as dynamic programming (e.g., Held-Karp algorithm), branch and bound, genetic algorithms, and approximation algorithms.

8. Can the Hamiltonian path approach be used to solve the TSP problem directly?

Answer: The Hamiltonian path approach is a step towards solving TSP, but the standard TSP usually requires finding a Hamiltonian cycle (a path that returns to the starting point). The brute-force approach provided here does not account for returning to the start city but can be adapted to find the Hamiltonian cycle if required.

9. What is an adjacency matrix and how is it used in the TSP implementation?

Answer: An adjacency matrix is a square matrix used to represent a graph, where the element at row iii and column j represents the weight of the edge between vertex iii and vertex j. In the TSP implementation, the adjacency matrix is used to look up the weight of edges when calculating the path length.

10. What are some practical applications of the TSP?

Answer: TSP has practical applications in logistics (e.g., optimizing delivery routes), manufacturing (e.g., minimizing tool movement), circuit design (e.g., designing electronic circuits), and various other fields requiring optimization of routes or sequences.

LAB 11: Write a program to find the maximum number of edge-disjoint paths between two vertices. Use the Edmonds-Karp algorithm, an implementation of the Ford Fulkerson method for computing the maximum flow in a flow network.

11.1 Objectives:

Use python

- to implement the Edmonds-Karp algorithm.
- to calculate the maximum number of edge-disjoint paths from source to sink.
- to find the flow values on edges.

11.2 The Edmonds-Karp algorithm, an implementation of the Ford Fulkerson method for computing the maximum flow in a flow network.

The Edmonds-Karp algorithm is a specific implementation of the Ford-Fulkerson method for computing the maximum flow in a flow network. It uses Breadth-First Search (BFS) to find augmenting paths, ensuring that the algorithm always finds the shortest augmenting path in terms of the number of edges. This results in a more predictable and efficient performance compared to other methods.

Flow of Edmonds-Karp Algorithm:

1. **Initialize Flow**: Start with zero flow in all edges.
2. **Find Augmenting Path**: Use BFS to find the shortest path from the source to the sink in the residual graph.
3. **Augment Flow**: Increase the flow along the found path by the minimum residual capacity of the path.
4. **Update Residual Capacities**: Adjust the residual capacities of the edges and reverse edges.
5. **Repeat**: Continue finding augmenting paths and updating flows until no more augmenting paths can be found. The maximum flow is the total flow from the source to the sink.

11.2.1 Implement the Edmonds-Karp algorithm to find the maximum number of edge-disjoint paths between two vertices in a flow network.

```
from collections import deque, defaultdict

class EdmondsKarp:
    def __init__(self, n):
        self.size = n
        self.capacity = defaultdict(lambda: defaultdict(int))
        self.flow = defaultdict(lambda: defaultdict(int))
        self.graph = defaultdict(list)

    def add_edge(self, u, v, cap):
        self.capacity[u][v] = cap
        self.graph[u].append(v)
        self.graph[v].append(u)  # Add reverse edge for residual capacity

    def bfs(self, source, sink, parent):
        visited = set()
        queue = deque([source])
        visited.add(source)

        while queue:
            u = queue.popleft()

            if u == sink:
                return True

            for v in self.graph[u]:
                if v not in visited and self.capacity[u][v] > self.flow[u][v]:
                    parent[v] = u
                    visited.add(v)
                    queue.append(v)
                    if v == sink:
                        return True

        return False

    def edmonds_karp(self, source, sink):
        max_flow = 0
        parent = {}

        while self.bfs(source, sink, parent):
            path_flow = float('Inf')
            s = sink

            while s != source:
                path_flow = min(path_flow, self.capacity[parent[s]][s] - self.flow[parent[s]][s])
                s = parent[s]

            v = sink
            while v != source:
                u = parent[v]
                self.flow[u][v] += path_flow
                self.flow[v][u] -= path_flow
                v = parent[v]

            max_flow += path_flow
        return max_flow

# Example usage
if __name__ == "__main__":
    n = 6  # Number of nodes
    ek = EdmondsKarp(n)  # Create an instance of EdmondsKarp

    # Add edges with capacity 1 to find edge-disjoint paths
    ek.add_edge(0, 1, 1)
    ek.add_edge(0, 2, 1)
    ek.add_edge(1, 2, 1)
    ek.add_edge(1, 3, 1)
    ek.add_edge(2, 4, 1)
    ek.add_edge(3, 4, 1)
    ek.add_edge(3, 5, 1)
    ek.add_edge(4, 5, 1)

    source = 0
    sink = 5
    max_edge_disjoint_paths = ek.edmonds_karp(source, sink)
    print(f"The maximum number of edge-disjoint paths from source {source} to sink {sink} is
{max_edge_disjoint_paths}.")
```

OUTPUT:

```
The maximum number of edge-disjoint paths from source 0 to sink 5 is 2.
```

11.2.2 Find the maximum flow in a flow network using the Edmonds-Karp algorithm.

```
import networkx as nx

def edmonds_karp(graph, source, sink):
    """Find the maximum flow in a flow network using the Edmonds-Karp algorithm.

    Args:
    - graph (nx.DiGraph): The directed graph with capacities as edge attributes.
    - source (node): The source node.
    - sink (node): The sink node.

    Returns:
    - max_flow (float): The maximum flow value from source to sink.
    - flow_dict (dict): A dictionary with flow values for each edge.
    """
    # Initialize flow to zero for all edges
    flow_value, flow_dict = nx.maximum_flow(graph, source, sink,
flow_func=nx.algorithms.flow.edmonds_karp)
    return flow_value, flow_dict
```

```
def main():
    # Create a directed graph with capacities
    G = nx.DiGraph()

    # Add edges along with capacities
    edges = [
        ('s', 'a', 10),
        ('s', 'b', 5),
        ('a', 'b', 15),
        ('a', 't', 10),
        ('b', 't', 10)
    ]

    G.add_weighted_edges_from(edges, weight='capacity')

    # Define source and sink
    source = 's'
    sink = 't'

    # Compute maximum flow
    max_flow, flow_dict = edmonds_karp(G, source, sink)

    print(f"Maximum Flow: {max_flow}")
    print("Flow values on edges:")
    for u, v, data in G.edges(data=True):
        print(f"Edge {u} -> {v}: Flow = {flow_dict[u][v]}, Capacity =
{data['capacity']}")

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Maximum Flow: 15
Flow values on edges:
Edge s -> a: Flow = 10, Capacity = 10
Edge s -> b: Flow = 5, Capacity = 5
Edge a -> b: Flow = 0, Capacity = 15
Edge a -> t: Flow = 10, Capacity = 10
Edge b -> t: Flow = 5, Capacity = 10
```

11.3 Exercise:

1. Write a program to optimize routing in transportation systems to ensure multiple independent routes between key locations using Python.
2. Design networks to ensure multiple independent communication paths between key nodes for reliability and fault tolerance using Python.

Viva Questions and Answers

1. What is the Edmonds-Karp algorithm?

Answer: The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method for computing the maximum flow in a flow network. It uses Breadth-First Search (BFS) to find augmenting paths in the residual graph, ensuring a polynomial time complexity.

2. How does the Edmonds-Karp algorithm work?

Answer: The algorithm repeatedly finds augmenting paths from the source to the sink using BFS. It then updates the capacities of the edges along these paths and adds the flow to the total maximum flow. This process continues until no more augmenting paths can be found.

3. What is the purpose of the `bfs` function in the Edmonds-Karp algorithm?

Answer: The `bfs` function is used to find an augmenting path from the source to the sink in the residual graph. It updates the parent array to keep track of the path and checks if a path exists.

4. What is the `edmonds_karp` function used for?

Answer: The `edmonds_karp` function calculates the maximum flow from the source to the sink in the flow network. It uses the `bfs` function to find augmenting paths and updates the capacities accordingly.

5. How is the maximum flow related to the maximum number of edge-disjoint paths?

Answer: The maximum number of edge-disjoint paths between two vertices in a network is equal to the maximum flow from the source to the sink. This is due to the Max-Flow Min-Cut Theorem, which states that the maximum flow in a network equals the capacity of the minimum cut separating the source and sink.

6. What is the time complexity of the Edmonds-Karp algorithm?

Answer: The time complexity of the Edmonds-Karp algorithm is $O(VE^2)$, where V is the number of vertices and E is the number of edges. This is due to the BFS operation running in O(E) time and being performed O(V) times.

7. How does the `path_flow` variable contribute to the algorithm?

Answer: The `path_flow` variable represents the minimum capacity along the augmenting path found by BFS. This value is used to update the capacities of the edges and to add to the total maximum flow.

8. What is a residual graph, and how is it used in the algorithm?

Answer: A residual graph is a modified version of the original graph that reflects the remaining capacities of the edges after some flow has been sent. It is used to find augmenting paths and update edge capacities during the execution of the algorithm.

9. Can the Edmonds-Karp algorithm handle networks with negative capacities?

Answer: No, the Edmonds-Karp algorithm assumes non-negative capacities. Negative capacities are not supported because they would complicate the notion of flow and capacity in the network.

10. How can you modify the program to handle larger graphs or different types of inputs?

Answer: For larger graphs or different inputs, ensure the program is efficient in terms of memory and processing. Consider using more advanced flow algorithms, such as the Dinic's algorithm, if performance becomes an issue. Additionally, ensure that the capacity matrix is correctly formatted and normalized for the given problem.

LAB 12: Using Menger’s theorem, find the minimum vertex cut between source and target.

12.1 Objectives:

Use python

- to find the minimum vertex cut between a source and target node in a graph using Menger's theorem

12.2 Menger's Theorem (Vertex Version):

The size of the minimum vertex cut between two vertices s and t in a graph is equal to the maximum number of vertex-disjoint paths between s and t.

To find the minimum vertex cut using Menger's theorem, we can follow these steps:

1. **Convert the Graph**: Create a flow network where each edge has capacity 1, and then use the maximum flow algorithm to find the maximum number of vertex-disjoint paths.
2. **Find the Maximum Flow**: Use an algorithm like Edmonds-Karp (a specific implementation of Ford-Fulkerson) to find the maximum flow in the network. This corresponds to the maximum number of vertex-disjoint paths.
3. **Derive the Minimum Vertex Cut**: Once we have the maximum flow, the minimum vertex cut can be derived from the residual flow network by identifying which vertices are reachable from the source in the residual graph.

12.2.1 Find the minimum vertex cut between source and target using Menger's theorem.

```
import networkx as nx

def minimum_vertex_cut(graph, source, target):
    """Find the minimum vertex cut between source and target.

    Args:
    - graph (nx.Graph): The undirected graph.
    - source (node): The source node.
    - target (node): The target node.

    Returns:
    - min_cut (set): The minimum vertex cut set.
    """
    # Convert undirected graph to a directed graph with capacity 1 on all edges
    flow_network = nx.DiGraph()
    for u, v in graph.edges:
        flow_network.add_edge(u, v, capacity=1)
        flow_network.add_edge(v, u, capacity=1)  # Add reverse edge for residual
capacity

    # Compute maximum flow using Edmonds-Karp
    flow_value, _ = nx.maximum_flow(flow_network, source, target,
flow_func=nx.algorithms.flow.edmonds_karp)

    # Find the minimum cut
    cut_value, partition = nx.minimum_cut(flow_network, source, target)
    reachable, non_reachable = partition

    # The minimum vertex cut is the set of nodes in the reachable set excluding the
source
    min_cut = set(reachable) - {source}

    return min_cut

def main():
    # Create an undirected graph
    G = nx.Graph()

    # Add edges
    edges = [
        ('s', 'a'),
        ('s', 'b'),
        ('a', 'b'),
        ('a', 'c'),
        ('b', 'c'),
        ('b', 'd'),
        ('c', 'd'),
        ('c', 't'),
        ('d', 't')
    ]

    G.add_edges_from(edges)

    # Define source and target
    source = 's'
    target = 't'

    # Compute the minimum vertex cut
    min_cut = minimum_vertex_cut(G, source, target)

    print(f"The minimum vertex cut between {source} and {target} is {min_cut}")

if __name__ == "__main__":
    main( )
```

OUTPUT:

```
The minimum vertex cut between s and t is {'d', 'b', 'c', 'a'}
```

12.3 Exercise:

1. Determine the minimum number of nodes (e.g., routers, switches) that need to be removed to disconnect a network using Python.

Viva Questions and Answers

1. What is Menger's theorem?

Answer: Menger's theorem states that in a graph, the minimum number of vertices (or edges) that need to be removed to disconnect a source vertex from a target vertex is equal to the maximum number of vertex-disjoint (or edge-disjoint) paths between the source and target.

2. How can Menger's theorem be used to find the minimum vertex cut in a graph?

Answer: To find the minimum vertex cut using Menger’s theorem, you compute the maximum number of vertex-disjoint paths between the source and target. This number is equal to the minimum number of vertices that need to be removed to disconnect the source from the target.

3. What is the Edmonds-Karp algorithm, and how does it relate to Menger's theorem?

Answer: The Edmonds-Karp algorithm is an implementation of the Ford-Fulkerson method for finding the maximum flow in a flow network. It uses BFS to find augmenting paths. This maximum flow value is equal to the maximum number of vertex-disjoint paths, which by Menger’s theorem, is also equal to the minimum vertex cut.

4. How does the `bfs` function work in the Edmonds-Karp algorithm?

Answer: The `bfs` function finds an augmenting path from the source to the sink in the residual graph using Breadth-First Search. It updates the parent array to track the path and determines if a path exists.

5. What is the purpose of the `edmonds_karp` function?

Answer: The `edmonds_karp` function calculates the maximum flow from the source to the sink in the flow network. It repeatedly finds augmenting paths and updates the capacities of the edges along these paths.

6. How is the minimum vertex cut determined from the residual graph?

Answer: The minimum vertex cut is found by identifying vertices reachable from the source in the residual graph but not reachable from the sink. The edges between these reachable and non-reachable vertices form the minimum vertex cut.

7.What is the significance of the maximum flow value?

Answer: The maximum flow value represents the maximum number of vertex-disjoint paths between the source and the target, and is also equal to the minimum number of vertices that need to be removed to disconnect the source from the target.

8.What is a residual graph, and how is it used in the algorithm?

Answer: A residual graph reflects the remaining capacities of the edges after some flow has been sent. It is used to find augmenting paths and update edge capacities during the execution of the maximum flow algorithm.

9.Can the algorithm handle graphs with negative capacities?

Answer: No, the algorithm assumes non-negative capacities. Negative capacities are not supported because they complicate the notion of flow and capacity in the network.

10. How can the program be adapted for larger graphs or different types of inputs?

Answer: For larger graphs, ensure efficient memory usage and consider using advanced algorithms or optimizations. For different types of inputs, validate and pre-process the capacity matrix to ensure it is correctly formatted for the algorithm.