25ALS040P: Discrete Mathematics

Assignment-1

1. Python Program for Set Operations

Problem Statement: Create a python program to perform following set operation: Creation of sets, Union, Intersection, Difference, Symmetric Difference, Set Cardinality, Powerset, Cartesian Product, and Set Complement.

Logic & Approach: This program demonstrates standard set operations using Python's built-in set data type and the itertools library.

- **Core Operations:** Union (|), Intersection (&), Difference (-), and Symmetric Difference (^) are performed using their intuitive operators.
- **Cardinality:** The len() function is used to find the number of elements in a set.
- **Complement:** The complement of a set A is defined as U - A, where U is the universal set.
- **Powerset:** A custom function generates the set of all subsets. It iterates from 0 to 2^n - 1, using the binary representation of each number to select elements for a subset.
- **Cartesian Product:** The itertools.product function is used for an efficient and standard implementation.

Python Code:

Python

```
import itertools


# 1. Creation of Sets

set_A = {1, 2, 3, 4, 5}

set_B = {4, 5, 6, 7, 8}

U = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} # Universal Set for Complement


print(f"Set A: {set_A}")

print(f"Set B: {set_B}")
```

```python
print(f"Universal Set U: {U}\n")
print("--- Set Operations ---")

# 2. Union: Elements in either A or B
union_set = set_A | set_B
print(f"Union (A U B): {union_set}")

# 3. Intersection: Elements in both A and B
intersection_set = set_A & set_B
print(f"Intersection (A ∩ B): {intersection_set}")

# 4. Difference: Elements in A but not in B
difference_set = set_A - set_B
print(f"Difference (A - B): {difference_set}")

# 5. Symmetric Difference: Elements in either A or B, but not both
symmetric_diff_set = set_A ^ set_B
print(f"Symmetric Difference (A Δ B): {symmetric_diff_set}")

# 6. Set Cardinality: Number of elements in the set
cardinality_A = len(set_A)
print(f"Cardinality of A (|A|): {cardinality_A}")

# 7. Powerset: The set of all subsets of A
def get_powerset(s):
    """Generates the powerset of a given set."""
    s_list = list(s)
    powerset = []
```

```
    # The number of subsets is 2^n
    num_subsets = 2 ** len(s_list)
    # Iterate through all possible binary numbers from 0 to 2^n - 1
    for i in range(num_subsets):
        subset = []
        # Check each bit of the binary number
        for j in range(len(s_list)):
            # If the j-th bit is set, include the j-th element
            if (i >> j) & 1:
                subset.append(s_list[j])
        powerset.append(frozenset(subset)) # Use frozenset for a set of sets
    return set(powerset)

# Note: Powerset is large, so we demonstrate with a smaller set
small_set = {1, 2, 3}
powerset_A = get_powerset(small_set)
print(f"Powerset of {small_set}: {powerset_A}")

# 8. Cartesian Product: The set of all ordered pairs (a, b)
# We use a smaller set for readability
cartesian_set_A = {'a', 'b'}
cartesian_set_B = {1, 2}
cartesian_product = set(itertools.product(cartesian_set_A, cartesian_set_B))
print(f"Cartesian Product of {{'a', 'b'}} x {{1, 2}}: {cartesian_product}")

# 9. Set Complement: Elements in U but not in A
complement_A = U - set_A
print(f"Complement of A (A'): {complement_A}")
```

Sample Output:

Set A: {1, 2, 3, 4, 5}

Set B: {4, 5, 6, 7, 8}

Universal Set U: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

--- Set Operations ---

Union (A U B): {1, 2, 3, 4, 5, 6, 7, 8}

Intersection (A ∩ B): {4, 5}

Difference (A - B): {1, 2, 3}

Symmetric Difference (A Δ B): {1, 2, 3, 6, 7, 8}

Cardinality of A (|A|): 5

Powerset of {1, 2, 3}: {frozenset({1, 2}), frozenset({1, 3}), frozenset({2, 3}), frozenset({2}), frozenset({1, 2, 3}), frozenset(), frozenset({3}), frozenset({1})}

Cartesian Product of {'a', 'b'} x {1, 2}: {('b', 1), ('a', 2), ('a', 1), ('b', 2)}

Complement of A (A'): {6, 7, 8, 9, 10}

---

2. Pigeonhole Principle Demonstration

Problem Statement: Find and demonstrate the Pigeonhole Principle in python programming.

Logic & Approach: The **Pigeonhole Principle** states that if you have **N** items (pigeons) to be placed into **M** containers (pigeonholes) and **N > M**, then at least one container must hold more than one item.

This program demonstrates the principle by:

1. Taking the number of pigeons and pigeonholes as input.
2. Checking if pigeons > pigeonholes.
3. If the condition is met, it explains the principle and calculates the minimum number of pigeons that must be in at least one hole using the formula: ceil(N / M).
4. It then runs a simple simulation by randomly assigning each pigeon to a hole and displays the final distribution, proving that at least one hole contains multiple pigeons.

Python Code:

Python

```python
import random
import math


def demonstrate_pigeonhole(pigeons, pigeonholes):
    """
    Demonstrates the Pigeonhole Principle with a simulation.
    """
    print(f"--- Pigeonhole Principle Demonstration ---")
    print(f"Number of Pigeons (items): {pigeons}")
    print(f"Number of Pigeonholes (containers): {pigeonholes}\n")

    if pigeons <= pigeonholes:
        print("The condition for the Pigeonhole Principle (pigeons > pigeonholes) is not met.")
        return

    print("Principle: Since there are more pigeons than pigeonholes,")
    print("at least one pigeonhole must contain more than one pigeon.\n")

    # Calculate the minimum number of pigeons in the most crowded hole
    min_in_one_hole = math.ceil(pigeons / pigeonholes)
    print(f"By the generalized principle, at least one hole must contain at least
{min_in_one_hole} pigeons.\n")

    # Simulation
    print("--- Simulation ---")
    # Create empty pigeonholes (a list of lists)
```

```
    holes = [[] for _ in range(pigeonholes)]

    # Randomly place each pigeon in a hole
    for pigeon_id in range(1, pigeons + 1):
        hole_number = random.randint(0, pigeonholes - 1)
        holes[hole_number].append(f"Pigeon-{pigeon_id}")

    # Display the results
    print("Distribution of pigeons in holes:")
    found_multiple = False
    for i, hole_content in enumerate(holes):
        count = len(hole_content)
        print(f"Hole {i+1}: {count} pigeons -> {hole_content}")
        if count > 1:
            found_multiple = True

    print("\nSimulation Result:")
    if found_multiple:
        print("As predicted, at least one pigeonhole was found containing more than one
pigeon.")
    else:
        # This case should not be reachable if pigeons > pigeonholes
        print("An unexpected error occurred in the simulation.")

# --- Main Execution ---
if __name__ == "__main__":
    # Example where the principle applies
    demonstrate_pigeonhole(pigeons=10, pigeonholes=9)
```

```
    print("\n" + "="*50 + "\n")
    # Example where it doesn't apply
    demonstrate_pigeonhole(pigeons=5, pigeonholes=5)
```

Sample Output:

--- Pigeonhole Principle Demonstration ---

Number of Pigeons (items): 10

Number of Pigeonholes (containers): 9

Principle: Since there are more pigeons than pigeonholes,

at least one pigeonhole must contain more than one pigeon.

By the generalized principle, at least one hole must contain at least 2 pigeons.

--- Simulation ---

Distribution of pigeons in holes:

Hole 1: 1 pigeons -> ['Pigeon-7']

Hole 2: 1 pigeons -> ['Pigeon-8']

Hole 3: 1 pigeons -> ['Pigeon-2']

Hole 4: 1 pigeons -> ['Pigeon-4']

Hole 5: 2 pigeons -> ['Pigeon-5', 'Pigeon-9']

Hole 6: 1 pigeons -> ['Pigeon-3']

Hole 7: 1 pigeons -> ['Pigeon-6']

Hole 8: 1 pigeons -> ['Pigeon-1']

Hole 9: 1 pigeons -> ['Pigeon-10']

Simulation Result:

As predicted, at least one pigeonhole was found containing more than one pigeon.

==

--- Pigeonhole Principle Demonstration ---

Number of Pigeons (items): 5

Number of Pigeonholes (containers): 5

The condition for the Pigeonhole Principle (pigeons > pigeonholes) is not met.

---

3. Inclusion-Exclusion Principle Implementation

Problem Statement: Create a python program including a function inclusion_exclusion that takes a list of sets as input. It uses bitwise operations to generate all possible subsets of the sets and calculates the intersection of each subset. By applying the inclusion-exclusion formula, it computes the sum of the lengths of the intersections with alternating signs. Finally, it computes the size of the union.

Logic & Approach: The **Inclusion-Exclusion Principle** is a counting technique to find the number of elements in the union of multiple sets. The formula for three sets A, B, C is: $|A \cup B \cup C| = |A| + |B| + |C| - (|A \cap B| + |A \cap C| + |B \cap C|) + |A \cap B \cap C|$

This program generalizes the formula for any number of sets.

1. It iterates through all non-empty subsets of the input list of sets. A number i from 1 to 2^n - 1 is used as a bitmask to select which sets to include in the current subset.
2. For each subset, it calculates their common intersection.
3. Based on the size of the subset of sets (e.g., intersection of 2 sets vs. 3 sets), it either adds or subtracts the size of this intersection from a running total. If the subset size is odd, it adds; if even, it subtracts.
4. The final total is the size of the union of all sets.

Python Code:

Python

```
def inclusion_exclusion(list_of_sets):
    """
    Calculates the size of the union of sets using the Inclusion-Exclusion Principle.
```

```
"""
num_sets = len(list_of_sets)
if num_sets == 0:
    return 0

total_union_size = 0

# Iterate through all non-empty subsets of the list_of_sets.
# We use a number from 1 to 2^n - 1 as a bitmask.
for i in range(1, 1 << num_sets):
    current_intersection = list_of_sets[0].copy() # Start with a copy of a set
    subset_size = 0

    # Build the intersection for the current subset of sets
    for j in range(num_sets):
        # If the j-th bit is set in i, this set is in our current subset
        if (i >> j) & 1:
            if subset_size == 0:
                 # This is the first set in our subset, so we copy it
                current_intersection = list_of_sets[j].copy()
            else:
                # Otherwise, we intersect with the running intersection
                current_intersection.intersection_update(list_of_sets[j])
            subset_size += 1

    # Apply the alternating sign based on the size of the subset
    if subset_size % 2 == 1: # Odd number of sets in intersection: Add
        total_union_size += len(current_intersection)
```

```
        else: # Even number of sets in intersection: Subtract
            total_union_size -= len(current_intersection)

    return total_union_size

# --- Main Execution ---
if __name__ == "__main__":
    # Example with three sets
    set1 = {1, 2, 3, 4}
    set2 = {3, 4, 5, 6}
    set3 = {2, 3, 6, 7}

    sets = [set1, set2, set3]

    print(f"Sets: {sets}")

    # Calculate using the function
    union_size_calculated = inclusion_exclusion(sets)
    print(f"Calculated union size (Inclusion-Exclusion): {union_size_calculated}")

    # Verify using Python's built-in union operator
    actual_union = set1.union(set2).union(set3)
    print(f"Actual union set: {actual_union}")
    print(f"Actual union size: {len(actual_union)}")

    # Verification
    if union_size_calculated == len(actual_union):
        print("\nResult is correct! ✔")
```

```
else:
    print("\nResult is incorrect. ❌")
```

Sample Output:

```
Sets: [{1, 2, 3, 4}, {3, 4, 5, 6}, {2, 3, 6, 7}]
Calculated union size (Inclusion-Exclusion): 7
Actual union set: {1, 2, 3, 4, 5, 6, 7}
Actual union size: 7

Result is correct! ✔
```