

## Part:B

### Practical:1

**AIM:** Write a PROLOG program to represent Facts. Make some queries based on the facts

#### Objective:

To write a Prolog program that represents facts and performs queries based on those facts, demonstrating the logic programming approach in artificial intelligence.

#### Theory:

Prolog (Programming in Logic) is a declarative programming language widely used in AI and computational linguistics. In Prolog, knowledge is represented in the form of facts, rules, and queries. A fact is a basic assertion about some world state. Prolog queries are used to extract information based on the facts and rules provided.

#### Code:

```
% Facts about family relationships
```

```
parent(john, mary). % John is a parent of Mary
```

```
parent(john, peter). % John is a parent of Peter
```

```
parent(mary, tom). % Mary is a parent of Tom
```

```
parent(peter, lisa). % Peter is a parent of Lisa
```

```
% Queries:
```

```
% Who are the children of John?
```

```
% ?- parent(john, Child).
```

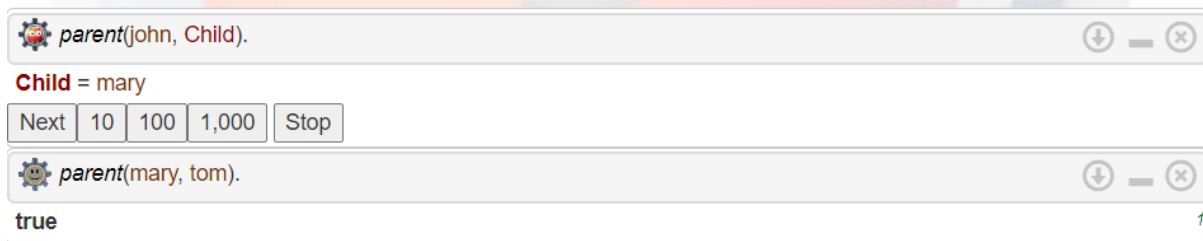
```
% Is Mary the parent of Tom?
```

```
Computer Engineering Department,
```

```
L. D. College of Engineering, Ahmedabad-15
```

```
% ?- parent(mary, tom).
```

## OUTPUT:



## Conclusion:

In this program, we represented basic family relationships using facts and queried the relationships. Prolog allows us to easily define relationships and extract information based on logical queries, showcasing the power of declarative programming.

## Practical:2

**AIM:** Write a PROLOG program to represent Rules. Make some queries based on the facts and rules.

### Objective:

To write a Prolog program that represents rules and facts, and perform queries to extract information based on both facts and rules.

### Theory:

Prolog programs consist of facts, rules, and queries. Rules allow Prolog to infer new information from existing facts. A rule is typically defined in the form of Head :- Body, meaning the "Head" is true if the "Body" is true. Prolog uses rules to deduce logical conclusions from facts.

### Code:

```
% Facts
```

```
parent(john, mary). % John is a parent of Mary
```

```
parent(john, peter). % John is a parent of Peter
```

```
parent(mary, tom). % Mary is a parent of Tom
```

```
parent(peter, lisa). % Peter is a parent of Lisa
```

```
% Rule: X is a grandparent of Y if X is a parent of Z and Z is a parent of Y
```

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

```
% Queries:
```

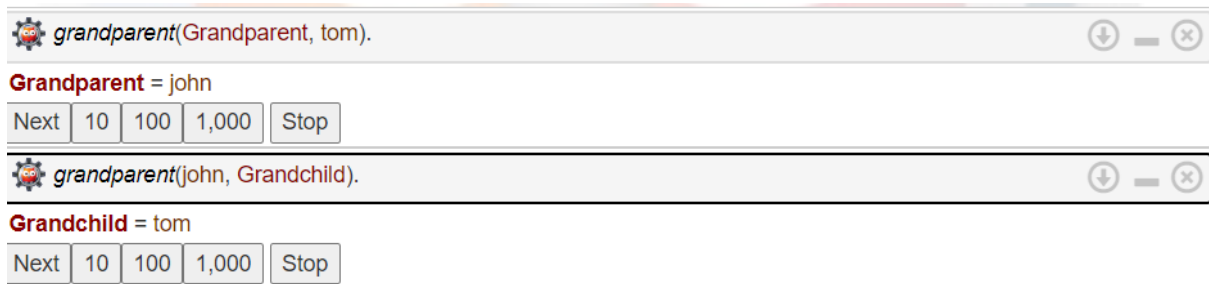
```
% Who are the grandparents of Tom?
```

```
% ?- grandparent(Grandparent, tom).
```

```
% Who are the grandchildren of John?
```

```
% ?- grandparent(john, Grandchild).
```

## OUTPUT:



The screenshot shows a Prolog interpreter window with two query sessions. The first session shows the query `grandparent(Grandparent, tom).` with the result `Grandparent = john`. The second session shows the query `grandparent(john, Grandchild).` with the result `Grandchild = tom`. Both sessions include a control bar with 'Next', '10', '100', '1,000', and 'Stop' buttons.

Query	Result
<code>grandparent(Grandparent, tom).</code>	<code>Grandparent = john</code>
<code>grandparent(john, Grandchild).</code>	<code>Grandchild = tom</code>

## Conclusion:

In this program, we represented rules and facts in Prolog to deduce relationships. By defining rules like "grandparent", we can infer indirect relationships. Prolog's logical deduction capabilities make it ideal for tasks involving complex relationships and reasoning.

## Practical:3

**AIM:** Write a PROLOG program to define the relations using following predicates:

- a) Define a predicate brother(X,Y) which holds iff X and Y are brothers.
- b) Define a predicate cousin(X,Y) which holds iff X and Y are cousins.
- c) Define a predicate grandson(X,Y) which holds iff X is a grandson of Y.
- d) Define a predicate descendent(X,Y) which holds iff X is a descendent of Y.

**Consider the following genealogical tree:**

father(a,b). father(a,c). father(b,d). father(b,e). father(c,f).

Say which answers, and in which order, are generated by your definitions for the following queries in Prolog:

?- brother(X, Y). ?- cousin(X, Y). ?- grandson(X, Y). ?- descendent(X, Y).

### Objective:

To write a Prolog program that defines family relationships using the predicates brother/2, cousin/2, grandson/2, and descendent/2, and perform queries to test the relations.

### Theory:

Prolog uses facts and rules to represent relationships and infer new knowledge through logical reasoning. Predicates like brother/2, cousin/2, grandson/2, and descendent/2 can be defined using both facts (such as family relations) and rules. These predicates are useful for modeling family trees and querying complex relationships in genealogical structures.

### Code:

```
% Facts representing family relationships
```

```
father(a, b).
```

```
father(a, c).
```

```
father(b, d).
```

```
father(b, e).
```

father(c, f).

% Rule: X and Y are brothers if they share the same father and are male

brother(X, Y) :- father(Z, X), father(Z, Y), X \= Y.

% Rule: X and Y are cousins if their fathers are brothers

cousin(X, Y) :- father(A, X), father(B, Y), brother(A, B).

% Rule: X is a grandson of Y if Y is a father of Z and Z is a father of X

grandson(X, Y) :- father(Y, Z), father(Z, X).

% Rule: X is a descendent of Y if Y is a father of Z, and Z is an ancestor of X

descendent(X, Y) :- father(Y, X).

descendent(X, Y) :- father(Y, Z), descendent(X, Z).

% Queries:

% ?- brother(X, Y).

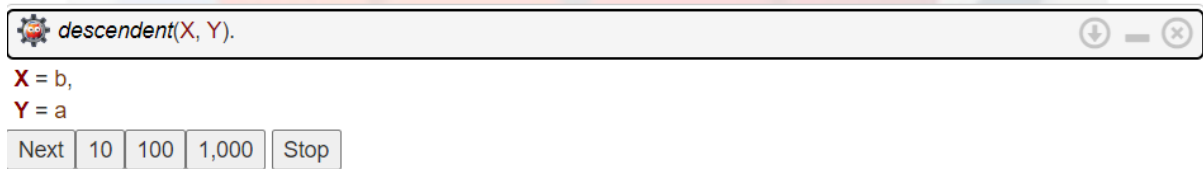
% ?- cousin(X, Y).

% ?- grandson(X, Y).

% ?- descendent(X, Y).

## OUTPUT:





## Conclusion:

In this program, we defined family relationships using Prolog predicates such as `brother/2`, `cousin/2`, `grandson/2`, and `descendent/2`. Prolog's logical inference allows us to query complex genealogical relationships based on a set of rules and facts. The program demonstrates how relationships can be deduced from simple facts using rules, making it a powerful tool for reasoning about family trees.

## Practical:4

**AIM:** Write a PROLOG program to perform addition, subtraction, multiplication and division of two numbers using arithmetic operators.

### Objective:

To write a Prolog program that performs addition, subtraction, multiplication, and division of two numbers using arithmetic operators.

### Theory:

Prolog is primarily a logic programming language, but it also supports basic arithmetic operations like addition (+), subtraction (-), multiplication (\*), and division (/). These operations can be performed using Prolog's `is/2` predicate, which evaluates arithmetic expressions and assigns the result to a variable.

### Code:

```
% Addition of two numbers
```

```
add(X, Y, Result) :- Result is X + Y.
```

```
% Subtraction of two numbers
```

```
subtract(X, Y, Result) :- Result is X - Y.
```

```
% Multiplication of two numbers
```

```
multiply(X, Y, Result) :- Result is X * Y.
```

```
% Division of two numbers
```

```
divide(X, Y, Result) :- Y \= 0, Result is X / Y.
```

```
% Queries:
```

```
% ?- add(10, 5, Result).
```

```
% ?- subtract(10, 5, Result).
```



% ?- multiply(10, 5, Result).

% ?- divide(10, 5, Result).

## OUTPUT:



The screenshot shows a Prolog interpreter window with four separate query boxes. Each box contains a query followed by the result. The queries are: `add(10, 5, Result).`, `subtract(10, 5, Result).`, `multiply(10, 5, Result).`, and `divide(10, 5, Result).`. The results are: `Result = 15`, `Result = 5`, `Result = 50`, and `Result = 2` respectively. Each box has a gear icon on the left and a close button on the right.

```
add(10, 5, Result).  
Result = 15  
subtract(10, 5, Result).  
Result = 5  
multiply(10, 5, Result).  
Result = 50  
divide(10, 5, Result).  
Result = 2
```

## Conclusion:

In this Prolog program, we used arithmetic operators to perform basic mathematical operations: addition, subtraction, multiplication, and division. By utilizing the `is/2` predicate, Prolog can evaluate arithmetic expressions and return the result, showcasing its ability to handle both logic-based queries and numerical computations.

## Practical:5

**AIM:** Write a PROLOG program to display the numbers from 1 to 10 by simulating the loop using recursion.

### Objective:

To write a Prolog program that simulates a loop using recursion to display numbers from 1 to 10.

### Theory:

Prolog does not have traditional loops like procedural languages. However, recursion can be used to simulate loops. Recursion involves a function calling itself with a modified argument until a base case is reached. This allows Prolog to iterate over values and perform repeated actions, such as displaying numbers.

### Code:

```
% Base case: Stop when N exceeds 10
```

```
print_numbers(11) :- !.
```

```
% Recursive case: Print the current number and call recursively with the next number
```

```
print_numbers(N) :-
```

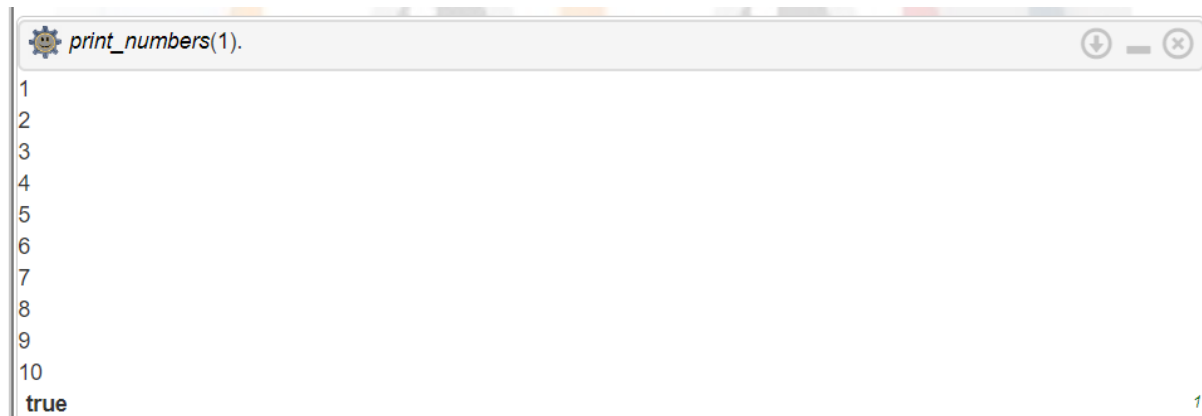
```
    write(N), nl,
```

```
    Next is N + 1,
```

```
    print_numbers(Next).
```

```
% Query:
```

```
% ?- print_numbers(1).
```

**OUTPUT:**

The screenshot shows a Prolog interpreter window titled 'print\_numbers(1)'. The window contains a list of numbers from 1 to 10, followed by the word 'true'. The numbers are listed vertically, and the 'true' is at the bottom. The window has a standard macOS-style title bar with a close button, a minimize button, and a maximize button.

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
true
```

**Conclusion:**

In this program, we used recursion to simulate a loop in Prolog. By defining a base case to stop the recursion and a recursive case to increment the number and print it, we successfully displayed numbers from 1 to 10. This demonstrates how recursion can be used to achieve repetitive tasks in Prolog, replacing traditional looping mechanisms.

## Practical:6

**AIM:** Write a PROLOG program to count number of elements in a list.

### Objective:

To write a Prolog program that counts the number of elements in a list using recursion.

### Theory:

In Prolog, lists are a fundamental data structure used to store elements. Recursion is often used to process lists, as it allows Prolog to traverse through each element until the end of the list. To count the number of elements, we recursively traverse the list, incrementing a counter until we reach an empty list.

### Code:

```
% Base case: The empty list has 0 elements
```

```
count_elements([], 0).
```

```
% Recursive case: Count the head and recursively count the tail
```

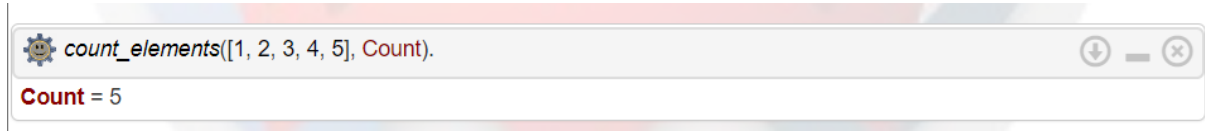
```
count_elements([_|Tail], Count) :-
```

```
    count_elements(Tail, TailCount),
```

```
    Count is TailCount + 1.
```

```
% Query:
```

```
% ?- count_elements([1, 2, 3, 4, 5], Count).
```

**OUTPUT:****Conclusion:**

This Prolog program uses recursion to count the number of elements in a list. By defining a base case for the empty list and a recursive case to count the remaining elements, we can traverse the entire list and determine its length. This demonstrates how Prolog handles list processing through recursion.

## Practical:7

**AIM:** Write a PROLOG program to count number of elements in a list. 7. Write a PROLOG program to perform following operations on a list:

- a. To insert an element into the list.
- b. To replace an element from the list.
- c. To delete an element from the list.

### Objective:

To write a Prolog program that counts the number of elements in a list and performs operations like inserting, replacing, and deleting elements from the list.

### Theory:

Prolog provides powerful recursion mechanisms to manipulate lists. Basic list operations like counting elements, inserting, replacing, and deleting elements can be done through pattern matching and recursion. These operations are key for handling data structures efficiently in Prolog.

### Code:

#### 1. Counting the number of elements in a list:

% Base case: The empty list has 0 elements

```
count_elements([], 0).
```

% Recursive case: Count the head and recursively count the tail

```
count_elements([_|Tail], Count) :-
```

```
    count_elements(Tail, TailCount),
```

```
    Count is TailCount + 1.
```

% Query: ?- count\_elements([1, 2, 3, 4, 5], Count).

## 2. Inserting an element into a list:

% Insert an element at the beginning of the list

insert(Element, List, [Element|List]).

% Query: ?- insert(6, [1, 2, 3, 4, 5], NewList).

## 3. Replacing an element in a list:

% Base case: Empty list remains unchanged

replace(\_, \_, [], []).

% If Head matches the element to be replaced, replace it

replace(Old, New, [Old|Tail], [New|NewTail]) :-

replace(Old, New, Tail, NewTail).

% If Head does not match, keep it and continue

replace(Old, New, [Head|Tail], [Head|NewTail]) :-

replace(Old, New, Tail, NewTail).

% Query: ?- replace(2, 9, [1, 2, 3, 4, 5], NewList).

## 4. Deleting an element from a list:

% Base case: Empty list remains unchanged

delete(\_, [], []).

% If Head matches the element to be deleted, skip it

delete(Element, [Element|Tail], NewTail) :-

delete(Element, Tail, NewTail).

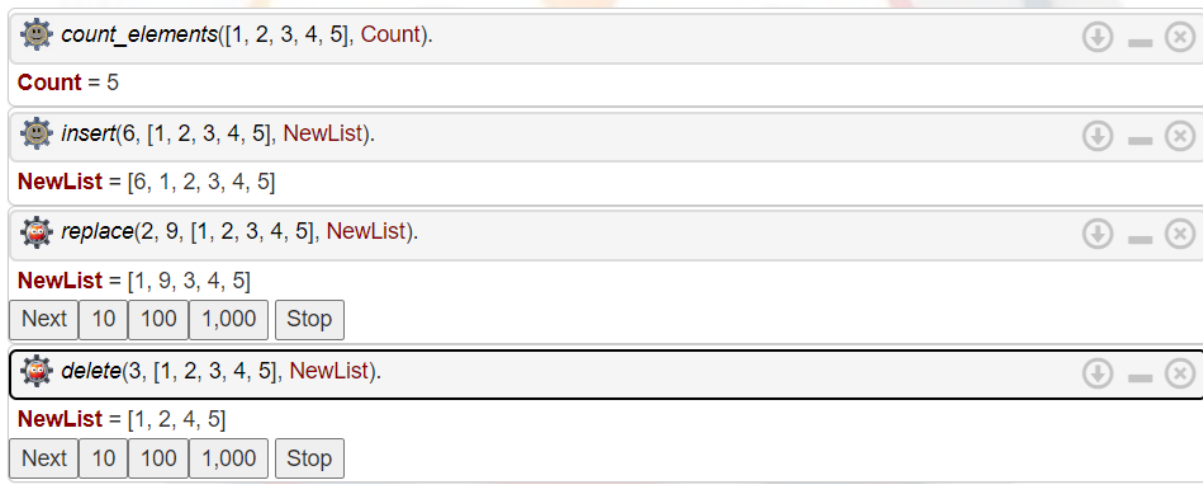
% If Head does not match, keep it and continue

```
delete(Element, [Head|Tail], [Head|NewTail]) :-
```

```
    delete(Element, Tail, NewTail).
```

```
% Query: ?- delete(3, [1, 2, 3, 4, 5], NewList).
```

## OUTPUT:



## Conclusion:

In this Prolog program, we performed basic list operations such as counting elements, inserting, replacing, and deleting elements from a list. These operations demonstrate how Prolog uses recursion and pattern matching to handle lists efficiently, making it a powerful tool for manipulating data structures.



## Practical:8

**AIM:** Write a PROLOG program to reverse the list.

### Objective:

To write a Prolog program that reverses a given list using recursion.

### Theory:

In Prolog, lists can be processed using recursion. To reverse a list, we can recursively move elements from the front of the list to the end of a new list. This recursive technique allows us to build the reversed list element by element until the original list is empty.

### Code:

% Base case: The reverse of an empty list is an empty list

```
reverse_list([], []).
```

% Recursive case: Reverse the tail of the list and append the head to the end

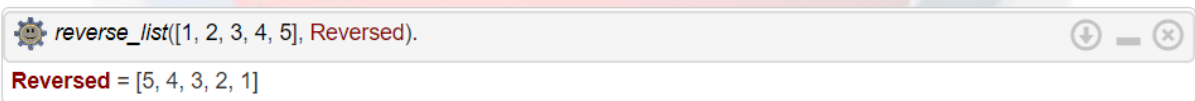
```
reverse_list([Head|Tail], Reversed) :-
```

```
    reverse_list(Tail, ReversedTail),
```

```
    append(ReversedTail, [Head], Reversed).
```

% Query: ?- reverse\_list([1, 2, 3, 4, 5], Reversed).

### OUTPUT:



```
reverse_list([1, 2, 3, 4, 5], Reversed).  
Reversed = [5, 4, 3, 2, 1]
```

### Conclusion:

In this program, we used recursion to reverse a list in Prolog. The base case handles the empty list, while the recursive case moves the head of the list to the end of a new reversed list. This demonstrates how Prolog can manipulate lists through recursive processing, making it effective for handling such tasks.

## Practical:9

**AIM:** Write a PROLOG program to demonstrate the use of CUT and FAIL predicate.

### Objective:

To write a Prolog program demonstrating the use of the CUT (!) and FAIL (fail) predicates for controlling backtracking.

### Theory:

In Prolog, the CUT (!) predicate is used to stop backtracking, committing Prolog to choices made up to that point in the computation. The FAIL predicate is used to force failure, which ensures that no further solutions are found for a query. Together, they are used for fine-grained control over Prolog's backtracking mechanism, enabling optimization of certain queries and behaviors.

- **CUT (!):** Prevents Prolog from backtracking beyond this point.
- **FAIL:** Forces the query to fail, ensuring no solutions are returned.

### Code:

```
% Example with CUT and FAIL
```

```
% Define some facts
```

```
likes(john, pizza).
```

```
likes(john, burger).
```

```
likes(mary, pasta).
```

```
likes(mary, pizza).
```

```
% Using CUT to stop backtracking
```

```
likes_only_pizza(Person) :-
```

```
    likes(Person, pizza), !.
```

% Using FAIL to force failure

likes\_nothing(Person) :-

likes(Person, \_), fail.

% Queries:

% ?- likes\_only\_pizza(john).

% ?- likes\_only\_pizza(mary).

% ?- likes\_nothing(john).

## OUTPUT:



## Conclusion:

This Prolog program demonstrates the use of the CUT and FAIL predicates. The CUT (!) is used to control backtracking, committing to the first valid solution and stopping further exploration. The FAIL predicate forces failure, ensuring no solutions are found. Together, these predicates allow for greater control over query execution and help manage Prolog's natural backtracking mechanism efficiently.

## Practical:10

**AIM:** Write a PROLOG program to solve the Tower of Hanoi problem.

### Objective:

To write a Prolog program to solve the Tower of Hanoi problem, which involves moving discs from one peg to another following specific rules.

### Theory:

The **Tower of Hanoi** is a classic recursive problem involving three pegs and a number of discs of different sizes. The objective is to move all the discs from the source peg to the destination peg, following these rules:

1. Only one disc can be moved at a time.
2. A disc can only be placed on top of a larger disc.
3. There are three pegs: source, auxiliary, and destination.

The solution to the problem can be expressed recursively by first moving  $n-1$  discs to the auxiliary peg, moving the largest disc to the destination, and finally moving the  $n-1$  discs from the auxiliary peg to the destination.

### Code:

% Base case: If there is only one disc, move it directly from Source to Destination

move(1, Source, Destination, \_) :-

write('Move disc 1 from '),

write(Source),

write(' to '),

write(Destination),

nl.

% Recursive case: Move  $n-1$  discs to Auxiliary, move  $n$ th disc to Destination, then move  $n-1$  discs from Auxiliary to Destination

move(N, Source, Destination, Auxiliary) :-

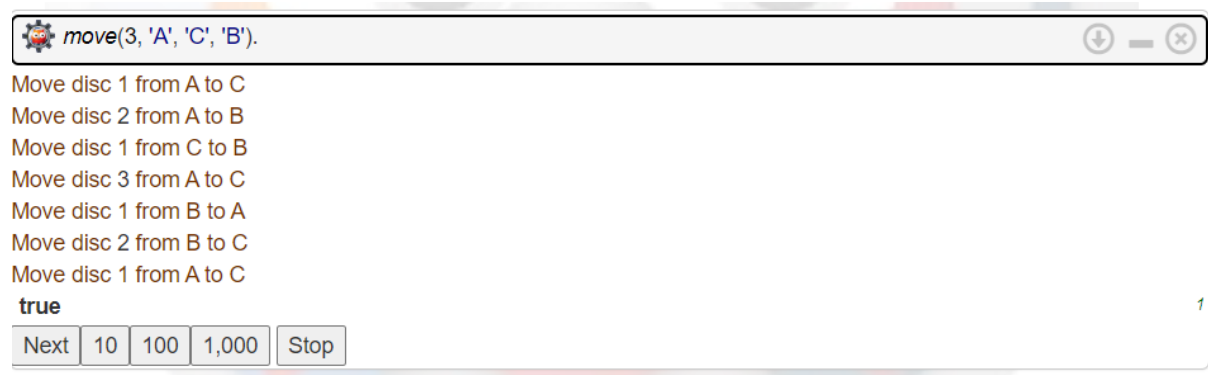
Computer Engineering Department,

L. D. College of Engineering, Ahmedabad-15

```
N > 1,  
M is N - 1,  
move(M, Source, Auxiliary, Destination),  
write('Move disc '),  
write(N),  
write(' from '),  
write(Source),  
write(' to '),  
write(Destination),  
nl,  
move(M, Auxiliary, Destination, Source).
```

% Query: ?- move(3, 'A', 'C', 'B').

## OUTPUT:



The screenshot shows a Prolog interpreter window with the title bar `move(3, 'A', 'C', 'B').`. The main text area displays the following output:

```
Move disc 1 from A to C  
Move disc 2 from A to B  
Move disc 1 from C to B  
Move disc 3 from A to C  
Move disc 1 from B to A  
Move disc 2 from B to C  
Move disc 1 from A to C  
true
```

At the bottom of the window, there is a control bar with buttons for `Next`, `10`, `100`, `1,000`, and `Stop`. A small page number `1` is visible in the bottom right corner of the text area.

## Conclusion:

The Tower of Hanoi problem is solved using recursion in Prolog. By breaking down the problem into smaller subproblems and recursively moving the discs, we efficiently solve it with a minimal number of moves. This program demonstrates how Prolog handles recursive algorithms and manages the complexity of such problems through elegant logic-based solutions.

## Practical:11

**AIM:** Write a PROLOG program to solve the N-Queens problem.

### Objective:

To write a Prolog program to solve the **N-Queens** problem, which involves placing N queens on an N×N chessboard such that no two queens can attack each other.

### Theory:

The **N-Queens problem** is a classical combinatorial problem in which N queens must be placed on an N×N chessboard so that no two queens threaten each other. This means:

- No two queens should be placed on the same row.
- No two queens should be placed on the same column.
- No two queens should be placed on the same diagonal.

The solution uses backtracking, where each queen is placed on the board one by one in a valid position. If no valid position is found for a queen, backtracking occurs to reposition the previous queen.

### Code:

```
% Check if the queen placement is safe from attacks
```

```
safe(_, []).
```

```
safe(X/Y, [X1/Y1 | Rest]) :-
```

```
    Y \= Y1,                % Ensure different rows
```

```
    abs(Y1 - Y) \= abs(X1 - X), % Ensure not on the same diagonal
```

```
    safe(X/Y, Rest).
```

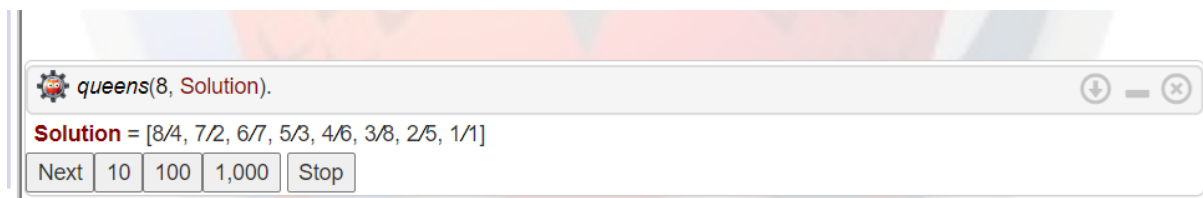
```
% Recursive function to place queens on the board
```

```
queens(N, Solution) :-
```

```
    queens(N, [], Solution).
```

```
% Helper function for placing queens  
queens(0, Solution, Solution). % Base case: no queens left to place  
queens(N, PartialSolution, Solution) :-  
    N > 0,  
    N1 is N - 1,  
    queens(N1, PartialSolution, NewPartialSolution),  
    between(1, 8, Y), % Generate Y (row) values between 1 and N  
    safe(N/Y, NewPartialSolution), % Check if the queen can be placed at (N, Y)  
    Solution = [N/Y | NewPartialSolution]. % Add the queen to the solution
```

## OUTPUT:



## Conclusion:

This Prolog program successfully solves the N-Queens problem by placing queens on an N×N chessboard using backtracking. The safe/2 predicate ensures no queens can attack each other, while the queens/2 predicate places the queens in valid positions. This program showcases Prolog's ability to efficiently handle combinatorial search problems through recursion and backtracking.

## Practical:12

**AIM:** Write a PROLOG program to solve the Travelling Salesman problem.

### Objective:

The objective of this program is to solve the **Travelling Salesman Problem (TSP)** using Prolog. The TSP aims to find the shortest possible route that visits a set of cities exactly once and returns to the starting point.

### Theory:

The Travelling Salesman Problem is a well-known optimization problem in computer science. Given a list of cities and distances between them, the goal is to determine the shortest path that visits each city once and returns to the starting city. The problem is NP-hard, meaning that as the number of cities increases, the complexity of finding the optimal solution grows exponentially. In this Prolog program, we will use **permutation** to generate all possible routes and then calculate the distance for each route, selecting the shortest one.

### Code:

```
% Facts representing the distances between cities
```

```
distance(a, b, 10).
```

```
distance(a, c, 15).
```

```
distance(a, d, 20).
```

```
distance(b, c, 35).
```

```
distance(b, d, 25).
```

```
distance(c, d, 30).
```

```
distance(b, a, 10).
```

```
distance(c, a, 15).
```

```
distance(d, a, 20).
```

```
distance(c, b, 35).
```

```
distance(d, b, 25).
```

```
distance(d, c, 30).
```



% Find the shortest path

tsp(Path, Cost) :-

findall((P, C), (permute([a, b, c, d], P), calculate\_cost(P, C)), Paths),

find\_min(Paths, (Path, Cost)).

% Permutation to generate all possible paths

permute([], []).

permute(List, [H | Perm]) :-

select(H, List, Rest),

permute(Rest, Perm).

% Calculate the total cost of a route

calculate\_cost([H | T], Cost) :-

calculate\_cost(T, H, 0, Cost).

calculate\_cost([H1 | T], H, Acc, Cost) :-

distance(H, H1, D),

Acc1 is Acc + D,

calculate\_cost(T, H1, Acc1, Cost).

calculate\_cost([], Last, Acc, Cost) :-

distance(Last, a, D), % Return to the starting point

Cost is Acc + D.

% Find the path with the minimum cost

find\_min([H | T], Min) :-

find\_min(T, H, Min).

find\_min([], Min, Min).

find\_min([(P, C) | T], (\_, Cmin), Min) :-

C < Cmin,

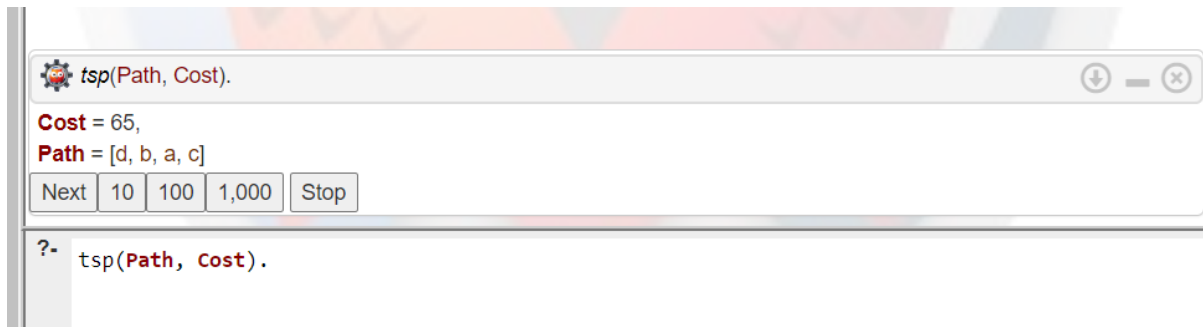
find\_min(T, (P, C), Min).

```
find_min([(_ , C)|T], (Pmin, Cmin), Min) :-
```

```
    C >= Cmin,
```

```
    find_min(T, (Pmin, Cmin), Min).
```

## OUTPUT:



## Conclusion:

The Prolog program successfully solves the **Travelling Salesman Problem** by generating all possible routes (permutations of cities) and calculating the total distance for each route. The program then selects the route with the shortest distance. Since Prolog uses recursion and backtracking, it explores all possible routes, ensuring that the optimal solution is found, but it may not be efficient for larger problems due to the computational complexity of permutation generation.