Number of regions founded by $n$ hyperplanes in $d$-dim passing through origin is given by the following recurrence relation

$$R_{n, d} = R_{n-1, d} + R_{n-1, d-1}$$

we use generating function as an operating function

Boundary condition:

$$R_{1, d} = 2 \quad \text{1 hyperplane in d-dim}$$
$$R_{n, 1} = 2 \quad \text{n hyperplanes in 1-dim, Reduce to n points thru origin}$$

The generating function is

$$f(x, y) = \sum_{n=1}^{\infty} \sum_{d=1}^{\infty} R_{n, d} \cdot x^n y^d$$
From the recurrence relation we have,

\[ R_{n, d} - R_{n-1, d} - R_{n-1, d-1} = 0 \]

\( R_{n-1,d} \) corresponds to ‘shifting’ \( n \) by 1 place, \( \Rightarrow \) multiplication by \( x \)

\( R_{n-1,d-1} \) corresponds to ‘shifting’ \( n \) and \( d \) by 1 place \( \Rightarrow \) multiplication by \( xy \)

On expanding \( f(x,y) \) we get

\[
f(x, y) = R_{1,1} \cdot xy + R_{1,2} \cdot x y^2 + R_{1,3} \cdot x y^3 + \ldots + R_{1,d} \cdot x y^d + \ldots \infty
\]

\[ + R_{2,1} \cdot x^2 y + R_{2,2} \cdot x^2 y^2 + R_{2,3} \cdot x^2 y^3 + \ldots + R_{2,d} \cdot x^2 y^d + \ldots \infty
\]

\[ \ldots \]

\[ + R_{n,1} \cdot x^n y + R_{n,2} \cdot x^n y^2 + R_{n,3} \cdot x^n y^3 + \ldots + R_{n,d} \cdot x^n y^d + \ldots \infty \]
\[ f(x, y) = \sum_{n=1}^{\infty} \sum_{d=1}^{\infty} R_{n,d} \cdot x^n y^d \]

\[ x \cdot f(x, y) = \sum_{n=1}^{\infty} \sum_{d=1}^{\infty} R_{n,d} \cdot x^{n+1} y^d = \sum_{n=2}^{\infty} \sum_{d=1}^{\infty} R_{n-1,d} \cdot x^n y^d \]

\[ xy \cdot f(x, y) = \sum_{n=1}^{\infty} \sum_{d=1}^{\infty} R_{n,d} \cdot x^{n+1} y^{d+1} = \sum_{n=2}^{\infty} \sum_{d=2}^{\infty} R_{n-1,d-1} \cdot x^n y^d \]

\[ x \cdot f(x, y) = \sum_{n=2}^{\infty} \sum_{d=2}^{\infty} R_{n-1,d} \cdot x^n y^d + \sum_{n=2}^{\infty} R_{n-1,1} \cdot x^n y \]

\[ = \sum_{n=2}^{\infty} \sum_{d=2}^{\infty} R_{n-1,d} \cdot x^n y^d + 2 \cdot \sum_{n=2}^{\infty} x^n y \]
\[ f(x, y) = \sum_{n=1}^{\infty} \sum_{d=1}^{\infty} R_{n,d} \cdot x^n y^d \]

\[ = \sum_{n=2}^{\infty} \sum_{d=2}^{\infty} R_{n,d} \cdot x^n y^d + \sum_{d=1}^{\infty} R_{1,d} \cdot xy^d + \sum_{n=1}^{\infty} R_{n,1} \cdot x^n y - R_{1,1} \cdot xy \]

\[ = \sum_{n=2}^{\infty} \sum_{d=2}^{\infty} R_{n,d} \cdot x^n y^d + 2x \cdot \sum_{d=1}^{\infty} y^d + 2y \cdot \sum_{n=1}^{\infty} x^n - 2xy \]

After all this expansion,

\[ f(x, y) - x \cdot f(x, y) - xy \cdot f(x, y) \]

\[ = \sum_{n=2}^{\infty} \sum_{d=2}^{\infty} (R_{n,d} - R_{n-1,d} - R_{n-1,d-1}) x^n y^d \]

\[ + 2y \sum_{n=1}^{\infty} x^n - 2xy - 2y \sum_{n=2}^{\infty} x^n + 2x \sum_{d=1}^{\infty} y^d \]

\[ = 2x \sum_{d=1}^{\infty} y^d \]

since other two terms become zero
This implies

\[ [1 - x - xy]f(x, y) = 2x \sum_{d=1}^{\infty} y^d \]

\[ f(x, y) = \frac{1}{[1 - x(1 - y)]} \cdot 2x \sum_{d=1}^{\infty} y^d \]

\[ = 2x[y + y^2 + y^3 + \ldots + y^d + \ldots\infty] \]

\[ [1 + x(1 + y) + x^2(1 + y)^2 + \ldots + x^d (1 + y)^d + \ldots\infty] \]

also we have,

\[ f(x, y) = \sum_{n=1}^{\infty} \sum_{d=1}^{\infty} R_{n, d} \cdot x^n y^d \]

Comparing coefficients of each term in RHS we get,
Comparing co-efficients we get

\[ R_{n, d} = \sum_{i=0}^{d-1} C_i^{n-1} \]
For perceptron

- $n =$ no. of inputs
- $d = n + 1$
- $R_{n,d}$ comes out to be upper bounded by $O(2^{n^2})$
Prolog
Introduction

- PROgramming in LOGic
- Emphasis on *what* rather than *how*
A Typical Prolog program

Compute_length ([], 0).
Compute_length ([Head|Tail], Length):-
  Compute_length (Tail, Tail_length),
  Length is Tail_length + 1.

High level explanation:

The length of a list is 1 plus the length of the tail of the list, obtained by removing the first element of the list.

This is a declarative description of the computation.
Fundamentals

*(absolute basics for writing Prolog Programs)*
Facts

- John likes Mary
  - like(john,mary)
- Names of relationship and objects must begin with a lower-case letter.
- Relationship is written *first* (typically the *predicate* of the sentence).
- *Objects* are written separated by commas and are enclosed by a pair of round brackets.
- The full stop character ‘.’ must come at the end of a fact.
## More facts

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>valuable(gold)</td>
<td>Gold is valuable.</td>
</tr>
<tr>
<td>owns(john,gold)</td>
<td>John owns gold.</td>
</tr>
<tr>
<td>father(john,mary)</td>
<td>John is the father of Mary</td>
</tr>
<tr>
<td>gives (john,book,mary)</td>
<td>John gives the book to Mary</td>
</tr>
</tbody>
</table>
Questions

- *Questions* based on facts
- Answered by *matching*

Two facts *match* if their predicates are same (spelt the same way) and the arguments each are same.

- If matched, prolog answers *yes*, else *no*.
- *No* does not mean falsity.
Prolog does *theorem proving*

- When a question is asked, prolog tries to match *transitively*.
- When no match is found, answer is *no*.
- This means *not provable* from the given facts.
Variables

- Always begin with a capital letter
  - ?- likes (john, X).
  - ?- likes (john, Something).

- But not
  - ?- likes (john, something)
Example of usage of variable

Facts:
  
  \[ \text{likes(john,flowers).} \]
  
  \[ \text{likes(john,mary).} \]
  
  \[ \text{likes(paul,mary).} \]

Question:
  
  \[ ?- \text{likes(john,X)} \]

Answer:
  
  \[ X=\text{flowers} \text{ and wait} \]
  
  \;
  
  \;
  
  \;
  
  \;
  
  \;
  
  \;
  
  no
Conjunctions

- Use ‘,’ and pronounce it as *and.*

- Example
  - Facts:
    - likes(mary,food).
    - likes(mary,tea).
    - likes(john,tea).
    - likes(john,mary)
  - ?-
    - likes(mary,X),likes(john,X).
    - Meaning *is anything liked by Mary also liked by John?*
Backtracking (an inherent property of prolog programming)

1. First goal succeeds. $X=food$
2. Satisfy $likes(john,food)$
Backtracking (continued)

Returning to a marked place and trying to resatisfy is called Backtracking

likes(mary,X), likes(john,X)

likes(mary,food)
likes(mary,tea)
likes(john,tea)
likes(john,mary)

1. Second goal fails
2. Return to marked place and try to resatisfy the first goal
Backtracking (continued)

1. First goal succeeds again, $X=tea$
2. Attempt to satisfy the $likes(john,tea)$
Backtracking (continued)

1. Second goal also succeeds
2. Prolog notifies success and waits for a reply
Rules

- Statements about objects and their relationships
- Express
  - If-then conditions
    - I use an umbrella if there is a rain
    - use(i, umbrella) :- occur(rain).
  - Generalizations
    - All men are mortal
    - mortal(X) :- man(X).
  - Definitions
    - An animal is a bird if it has feathers
    - bird(X) :- animal(X), has_feather(X).
Syntax

- `<head> :- <body>`
- Read ‘:-’ as ‘if’.
- E.G.
  - `likes(john,X) :- likes(X,cricket).`
  - “John likes X if X likes cricket”.
  - i.e., “John likes anyone who likes cricket”.
- Rules always end with ‘.’.
Another Example

sister_of (X,Y):- female (X),
parents (X, M, F),
parents (Y, M, F).

X is a sister of Y is
X is a female and
X and Y have same parents
Question Answering in presence of rules

- Facts
  - male (ram).
  - male (shyam).
  - female (sita).
  - female (gita).
  - parents (shyam, gita, ram).
  - parents (sita, gita, ram).
Question Answering: Y/N type: *is sita the sister of shyam?*

?- sister_of (sita, shyam)

- female(sita)
- parents(sita,M,F)
- parents(shyam,M,F)
- parents(sita,gita,ram)
- parents(shyam,gita,ram)

success
Question Answering: wh-type: *whose sister is sita?*

?- ?- sister_of (sita, X)

female(sita)

parents(sita,M,F)

parents(Y,M,F)

parents(sita,gita,ram)

parents(Y,gita,ram)

parents(shyam,gita,ram)

Success

Y=shyam
Rules

- Statements about *objects* and their *relationships*

- Express
  - *If-then conditions*
    - I use an umbrella if there is a rain
    - `use(i, umbrella) :- occur(rain)`.

- *Generalizations*
  - All men are mortal
    - `mortal(X) :- man(X)`.

- *Definitions*
  - An animal is a bird if it has feathers
    - `bird(X) :- animal(X), has_feather(X)`.
Make and Break

Fundamental to Prolog
Prolog examples using making and breaking lists

% incrementing the elements of a list to produce another list
incr1([],[]).
incr1([H|T1],[H2|T2]) :- H2 is H1+1, incr1(T1,T2).

% appending two lists; (append(L1,L2,L3) is a built in function in Prolog)
append1([],L,L).
append1([H|L1],L2,[H|L3]) :- append1(L1,L2,L3).

% reverse of a list (reverse(L1,L2) is a built in function)
reverse1([],[]).
reverse1([H|T],L):- reverse1(T,L1),append1(L1,[H],L).
Remove duplicates

Problem: to remove duplicates from a list

rem_dup([],[]).
rem_dup([H|T],L) :- member(H,T), !, rem_dup(T,L).
rem_dup([H|T],[H|L1]) :- rem_dup(T,L1).

Note: The cut ! in the second clause needed, since after succeeding at member(H,T), the 3\textsuperscript{rd} clause should not be tried even if rem_dup(T,L) fails, which prolog will otherwise do.
Member (membership in a list)

member(X,[X|_]).
member(X,[_|L]):- member(X,L).
Union (lists contain unique elements)

union([],Z,Z).
union([X|Y],Z,W):-
  member(X,Z),!,union(Y,Z,W).
union([X|Y],Z,[X|W]):- union(Y,Z,W).
Intersection (lists contain unique elements)

intersection([],Z,[]).
intersection([X|Y],Z,[X|W]):-
    member(X,Z),!,intersection(Y,Z,W).
intersection([X|Y],Z,W):-
    intersection(Y,Z,W).
Prolog Programs are close to Natural Language

Important Prolog Predicate:

\[ \text{member}(e, L) /* \text{true if } e \text{ is an element of list } L \]
\[ \text{member}(e, [e|L1]). /* e \text{ is member of any list which it starts} \]

\[ \text{member}(e, [\_|L1]):- \text{member}(e, L1) /* \text{otherwise } e \text{ is member of a list if the tail of the list contains } e \]

Contrast this with:

\[ \text{P.T.O.} \]
Prolog Programs are close to Natural Language, C programs are not

```c
For (i=0; i<length(L); i++) {
    if (e == a[i])
        break(); /* e found in a[] */
}
If (i<length(L)) {
    success(e, a); /* print location where e appears in a[] */
} else
    failure();

What is i doing here? Is it natural to our thinking?
Machine should ascend to the level of man

- A prolog program is an example of reduced man-machine gap, unlike a C program
- That said, a very large number of programs far outnumbering prolog programs get written in C
- The demand of practicality many times incompatible with the elegance of ideality
- But the ideal should nevertheless be striven for
Prolog Program Flow, BackTracking and Cut

Controlling the program flow
Prolog’s computation

- Depth First Search
  - Pursues a goal till the end
- Conditional AND; falsity of any goal prevents satisfaction of further clauses.
- Conditional OR; satisfaction of any goal prevents further clauses being evaluated.
Control flow (top level)

Given

\[
g:~ a, b, c. \quad (1)
\]

\[
g:~ d, e, f; g. \quad (2)
\]

If prolog cannot satisfy (1), control will automatically fall through to (2).
Control Flow within a rule

Taking (1),

\[ g:\:- \ a, \ b, \ c. \]

If \( a \) succeeds, prolog will try to satisfy \( b \), succeeding which \( c \) will be tried.

For ANDed clauses, control flows forward till the ‘.’, iff the current clause is \textit{true}.

For ORed clauses, control flows forward till the ‘.’, iff the current clause evaluates to \textit{false}. 

What happens on failure

- REDO the immediately preceding goal.
Fundamental Principle of prolog programming

- Always place the more general rule AFTER a specific rule.
CUT

- Cut tells the system that
  
  **IF YOU HAVE COME THIS FAR**
  
  **DO NOT BACKTRACK**
  
  **EVEN IF YOU FAIL SUBSEQUENTLY.**
  
  ‘CUT’ WRITTEN AS ‘!’ ALWAYS SUCCEEDS.
Fail

- This predicate always fails.
- *Cut* and *Fail* combination is used to produce negation.
- Since the LHS of the neck cannot contain any operator, $A \rightarrow \neg B$ is implemented as

\[ B :- A, !, Fail. \]