Algoritmo das N-Rainhas

Pedro Loes Daniel Rocha da Silva Pedro Toledo

30/06/2021

Contents

Introdução

- O objetivo deste projeto foi desenvolver um programa para resolver o problema das N-Rainhas utilizando linguagens de 3 paradigmas: imperativo, lógico e funcional, para as quais foram escolhidas as linguagens Java, Prolog e Lisp, respectivamente. Este relatório apresenta a descrição do problema, explica os algoritmo desenvolvidos, documenta os códigos e exemplifica o uso dos programas.

Problema N-Rainhas

- O problema consiste em posicionar um número N de rainhas em um tabuleiro de xadrez $X_{n,n}$.
- A peça rainha pode ser movimentada em um número ilimitado de casas nas linhas, colunas ou diagonais.
- Uma posição é segura se não está na mesma linha, coluna ou diagonal de posições ocupadas.

Java

Construção do Algoritmo

- O programa consiste em um algoritmo que recebe uma entrada do usuário e retorna todos os possíveis posicionamentos seguros das rainhas no tabuleiro $X_{n,n}$.
- O algoritmo foi desenvolvido com a implementação de 4 funções e 2 classes:

  1. <**verifique**>

  – Verifica se uma posição é segura.

  3. <**posicione**>

  – Executa laço recursivo para iterar sobre o espaço de busca.

  3. <**imprima**>

  – Imprime soluções encontradas pelo programa.

  4. <**nRainhas**>

  – Executa a função posicione na classe NRainhas.

  5. <**inteiroNaoPositivo**>

  – Extende a classe exceção.

  6. <**NRainhas**>

  – Classe principal de que recebe input do usuário e executa o programa.

- O funcionamento do algoritmo consiste em iterar sobre as linhas e colunas do tabuleiro $X_{n,n}$ procurando uma posição segura para cada rainha.
- O tabuleiro foi representado por um arranjo vetor X_{n} onde as colunas são as posições do vetor e as linhas os valores.
- A verificação de segurança para linhas é realizada comparando os valores do vetor X_n. A verificação de segurança nas diagonais é realizada comparando índices e valores do vetor X_n.
- Caso uma posição selecionada inviabilize o posicionamento de qualquer rainha posterior, o algoritmo retorna e reposiciona as rainhas anteriores de forma recursiva até encontrar uma solução onde as **n** rainhas estarão seguras.

1. Função Verifique

- A primeira função denominada <**verifique**>, verifica se uma posição em uma determinada casa do tabuleiro é segura. As linhas, colunas e diagonais são inspecionadas e a função retorna verdadeiro se as condições de segurança da posição forem satisfeitas.
- A função recebe como parâmetros:
  1. Um arranjo de inteiros <**tabuleiro**> que representa o tabuleiro X_n do problema.
  2. Número inteiro coluna que representa as colunas ou rainhas.
- O laço itera sobre todas as colunas do tabuleiro verificando 2 condições:
  1. Se a rainha está posicionada na mesma linha que as rainhas anteriores.
  2. Se a rainha está posicionada na mesma diagonal que as rainhas anteriores.
- Se as condições retornarem o valor verdadeiro, ou seja, existem duas rainhas na mesma linha ou diagonal, o programa retorna falso, ou seja, a nova posição não é segura. Caso contrário, a posição verificada é segura e a função retorna o valor verdadeiro.

```
// Declara função de verificação privada com retorno boleano
private static boolean verifique(int[] tabuleiro, int coluna) {

  // Laço para verificar linha
  for (int i = 0; i < coluna; i++) {

    // Verfica condição de linha
    if (tabuleiro[i] == tabuleiro[coluna])

      // Se verdadeiro retorna falso ou posição não segura
      return false;

    // Verfica condição de diagonal
    if ((coluna - i) == Math.abs(tabuleiro[coluna] - tabuleiro[i])) {

      // Se verdadeiro retorna falso ou posição não segura
      return false;
    }
  }

  // Retorna verdadeiro se a posição é segura
  return true;
}
```

2. Função Posicione

- A segunda função denominada <**posicione**>, itera sobre o tabuleiro de forma recursiva.
- A função recebe como parâmetros:
  1. O arranjo vetor de inteiros <**tabuleiro**> X_n.
  2. O inteiro <**coluna**> que representa a coluna da vez.
  3. O inteiro <**n**> que representa o máximo de rainhas.
- O laço itera sobre as colunas executando 3 tarefas:

1. Popula o arranjo <**tabuleiro**>.

   – A posição do arranjo <**tabuleiro**> recebe a linha segura para posicionamento.

2. Verifica se a posição (coluna, linha) é segura.

   – Executa a função verifique com parâmetros <**tabuleiro**> e <**coluna**> da vez.

3. Executa <**posicione**> de forma recursiva.

   – Caso a condição retornar verdadeiro, ou seja, posição segura, a função posicione é executada de forma recursiva alterando a coluna da vez para a próxima coluna.

```
// Declare função privada posicione com retorno vazio
private static void posicione(int[] tabuleiro, int coluna, int n) {

  // Condição de parada considerando o tamanho do problema
  if (coluna == n) {

    // Se verdadeiro imprime o tabuleiro
    imprima(tabuleiro);

    // Retorna vazio
    return;
  }

  // Laço para posicionar as rainhas
  for (int i = 0; i < n; i++) {

    // Popula rainha da vez para teste
    tabuleiro[coluna] = i;

    // Condição de verificação da posição
    if (verifique(tabuleiro, coluna)) {

      // Se verdadeiro popula tabuleiro
      posicione(tabuleiro, coluna + 1, n);
    }
  }
}
```

3. Função Imprima

- A terceira função denominada <**imprima**>, imprime os resultados do algoritmo no formato de um tabuleiro.
- Recebe como parâmetro de entrada o arranjo de inteiros <**tabuleiro**> da solução do problema.
- Itera sobre o tabuleiro imprimindo **'R'** para posição com rainha e **'.'** para posição vazia.
- Os tabuleiros foram separados por uma sequência de hífens **'- - - - - - - - - -'**.
- A condição de tabuleiro é verificada em cada impressão para evitar impressão de tabuleiros com entradas inválidas.

```java
// Declara função imprima estatica privada com retorno vazio
private static void imprima(int[] tabuleiro) {

  // Declara tamanho do tabuleiro
  int tamanho = tabuleiro.length;

  // Condição de Separação
  if(tamanho > 1)

    // Quebra de linha e divisão
    System.out.print("-----------------------------------");
    System.out.print("\n");

  // Laços para impressão do tabuleiro
  for (int i = 0; i < tamanho; i++) {
    for (int valor : tabuleiro) {

      // Condição de Rainhas
      if (valor == i && tamanho > 1)

        // Imprime rainha
        System.out.print("R\t");

      else if(valor != i && tamanho > 1)

        // Imprime posição vazia
        System.out.print(".\t");
    }

    // Condição de quebra de linha
    if(tamanho > 1)

      // Imprime quebra de linha e separador
      System.out.print("\n");

  }

}
```

4. Função nRainhas

- A função <**n_rainhas**> foi construída para gerar o arranjo de inteiros tabuleiro X_n que será populado pela solução construída na execução da função posicione iniciada na coluna 0.

```java
// Declara função estática privada NRainhas
 private static void nRainhas(int n) {

   // Declara tabuleiro com dimensão de N_rainhas
   int [] tabuleiro = new int[n];

   // Posiciona rainhas no tabuleiro
```

```
    posicione(tabuleiro, 0, n);
  }
```

5. Classe inteiroNaoPositivo

- A biblioteca *java.utils* foi utilizada para extender a classe <**Exception**>.
- A classe <**interoNaoPositivo**> foi construída para gerar exceção se a entrada do usuário não é um número inteiro. O construtor <**this**> recebe o parâmetro <**numero**> com tipo inteiro que será verificado pela extensão da exceção.

```
// Importa biblioteca de utilidades para escanear entrada do usuário
import java.util.*;

// Declara nova classe de exceção
class inteiroNaoPositivo extends Exception {

    // Declara variável
    private int naoPositivo = 0;

    // Construtor com argumento de inteiro
    public inteiroNaoPositivo(int numero) {

        // Declara atributo número
        this.naoPositivo = numero;
    }
}
```

6. Classe NRainhas

- A classe principal NRainhas foi desenvolvida para controlar e executar o programa.
- Composta de uma função principal e uma declaração <**try**> bloco de teste.

1. Função Principal <**main**>:

– Recebe o argumento <**args**> que é uma estrutura de dados arranjo do tipo *String*:
– Declara inteiro <**numeroEntrada**> inicializado com valor 0.
– Declara inteiro <**n**> de entrada inicializado com valor 1.
– Declara objeto entrada do tipo <**Scanner**> para recuperar a escolha do usuário.

2. Bloco de Testes <**try**>:

– Imprime mensagem requisitando a escolha do usuário para determinar o número de rainhas.
– Aciona módulo próximo inteiro do objeto <**entrada**> tipo *Scanner*.
– Verifica se número de entrada é negativo:
  * Se verdadeiro lança exceção interiroNaoPositivo.
  * Se falso pula.
– Atribui número de entrada a variável n.
– Declarações <**catch**>:
  * 1º <**catch**> imprime mensagem de erros se a entrada é negativa e reinicia o programa.

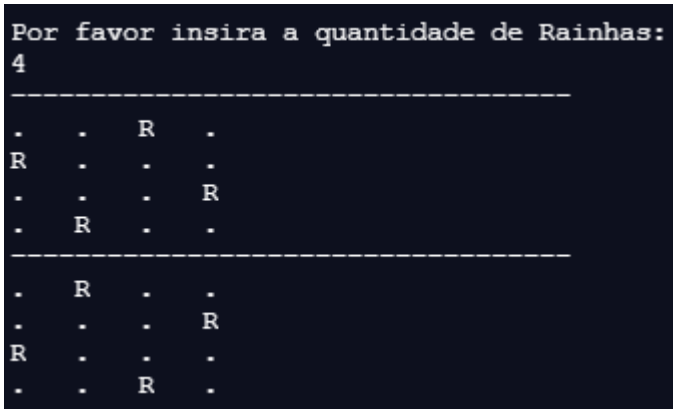
    - 2º <**catch**> imprime mensagem de erros se a entrada é diferente de inteiro e reinicia o programa.
  - A declaração <**finally**> permite executar código depois de <**try**>
    - Executa a função <**n_rainhas**> com parâmetro <**n**>.

```
// Declara classe publica principal
public class NRainhas {

  // Declara classe publica estática com retorno vazio
  public static void main(String[] args) {

    // Declara variáveis
    int numeroEntrada = 0; int n = 1;

    // Inicializa escaneador entrada
    Scanner entrada = new Scanner(System.in);

    // Teste de entrada
    try {

      // Solicita entrada do usuário
      System.out.println("Por favor insira a quantidade de Rainhas:");

      // Declara entrada
      numeroEntrada = entrada.nextInt();

      // Condição de negativo
      if (numeroEntrada < 0) {

        // Lança exceção de inteiro não negativo
        throw new inteiroNaoPositivo(numeroEntrada);
      }

      // Caso contrário declara variável n com entrada do usuário
      n = numeroEntrada;

    // Captura exceção de não positivo
    } catch (inteiroNaoPositivo e) {

      // Menssagem para usuário
      System.out.println("Entrada de inteiro negativo recusada!\n");
      main(args);

    // Captura exceção de entrada diferente de inteiro
    } catch (InputMismatchException e) {

      // Menssagem de entrada diferente de inteiro
      System.out.println("Entrada diferente de inteiro recusada!\n");
      main(args);

    // Retorno dos testes
    } finally {
```

```
        // Chama a função nRainhas com parâmetro n
        nRainhas(n);
    }
}
```

Exemplo

- Um exemplo da execução do programa é a resolução do problema com o parâmetro <**4**> para indicar a resolução do problema com 4 rainhas em um tabuleiro 4 x 4.
- Saída do Programa para o exemplo com 4 rainhas no tabuleiro 4 x 4:



Compilador para Execução

Abaixo é possível acessar o compilador online utilizado para a execução dos algoritmo

- A JVM do **replit** permite executar o programa no sítio https://replit.com
  - O Programa pode ser compilado e executado de 2 formas:
    1. Acionando o botão Run do menu superior central.
    2. Digitando os seguintes comando no Shell da janela principal da direita:
       * <**javac Main.java**>
       * <**java Main**>

Prolog

Construção do Algoritmo

- O algoritmo foi construindo implementando as condições e restrições necessárias ao problema. Inicialmente uma lista é construida contendo os seguintes valores: $N, N-1, N-2, ..., 1$ e em seguida essa lista é permutada até que uma permutação que atenda as condições e restrições seja encontrada. Em seguida o algoritmo busca outra solução. A solução é apresentada em formato de lista sendo que cada elemento corresponde a posição da rainha nas linhas do tabuleiro, de forma ordenada. A solução consiste em uma permutação da lista originalmente construída porque devemos ter, necessariamente, uma única rainha em cada linha ou coluna.

Restrições e Condições Implementadas

- As restrições e condições do problema em si foram implementadas utilizando fatos e regras na linguagem PROLOG. Cada um desses fatos e regras está descrito a seguir:

```
lista_rainhas(0, []).

lista_rainhas(N, [N|Linhas]) :-
    N > 0,
  N1 is N - 1,
  lista_rainhas(N1 , Linhas).
```

1. lista_rainhas\2 Nesta regra temos uma definição recursiva para uma regra com uma união disjunta representando o critério de parada da regra. Na avaliação da regra é verificado se N é maior que zero e, se isso for verdadeiro, o procedimento continua com $N - 1$ até atingir $N = 0$. O objeto "Linhas" que faz com que a regra seja verdadeira é utilizado pelo algoritmo posteriormente, esse objeto será (para N pertencente ao conjunto dos números naturais) da forma $N, N - 1, N - 2, ..., 1$.

```
rainhas(N, Posicao) :-
  lista_rainhas(N, Linhas),
  rainhas(Linhas , [], Posicao).
```

2. rainhas\2 Esta é a regra em que a consulta será feita. Para que as N rainhas nas posições *Posicao* sejam válidas (rainhas(N, Posicao)) temos que lista_rainhas(N, Linhas) precisa ser verdadeiro e o algoritmo encontrará o objeto "Linhas" que torna a condição verdadeira, como descrito anteriormente. Na condição "rainhas(Linhas , [], Posicao)" encontrará a Posicao que torna a condição verdadeira para o objeto "Linhas" passado.

```
select([X|Ys], X, Ys).

select([Y|Ys], X, [Y|Zs]) :- select(Ys , X, Zs).
```

3. select\3 Nesta regra temos a definição da condição de escolha de uma posição candidata. Essa regra será usada colocando no primeiro elemento a lista com todas as posições candidatas, no segundo elemento o número que representa a posição candidata e no terceiro elemento a nova lista candidata que é a lista original de candidatas sem o elemento candidato escolhido. Essa regra só é verdadeira quando o terceiro argumento é uma lista igual a lista do primeiro argumento a não ser pelo elemento do segundo argumento da regra.

```
posicaoSegura([], _Rainha , _Nb).

posicaoSegura([Y|Ys], Rainha , Nb) :-
```

```
    Rainha =\= Y + Nb ,
  Rainha =\= Y - Nb ,
  Nb1 is Nb + 1,
  posicaoSegura(Ys , Rainha , Nb1).
```

4. posicaoSegura\3 Esta regra verifica as condições e restrições fundamentais do problema que basicamente é a verificação da condição que nenhuma rainha posicionada consegue atacar qualquer outra rainha posicionada. Como a solução é buscada como permutações do objeto "Linhas" incial não é preciso verificar se há rainhas na mesma linha ou coluna que outra rainha, portanto é preciso verificar apenas se não há rainhas nas mesmas diagonais. A verificação é feita de forma recursiva começando com a rainha posicionada na linha anterior a linha da atual candidata e então a segunda linha anterior e assim sucessivamente. A verificação é feita somando e subtraindo Nb da posição das rainhas posicionadas nas linhas anteriores e verificando se o resultado dessas operações é diferente da posição da candidata, utilizando $Nb = 1$ para a primeira verificação, $Nb = 2$ para a segunda e assim por diante.

```
rainhas([], Posicao , Posicao).

rainhas(NaoPosicionada , Posicionada , Posicao) :-
    select(NaoPosicionada, R, NovaNaoPosicionada),
    posicaoSegura(Posicionada , R, 1),
    rainhas(NovaNaoPosicionada , [R|Posicionada], Posicao).
```

5. rainhas\3 Com esta regra o problema é finalmente solucionado verificando se as condições do "select" e "posicaoSegura" são verdadeiras para cada nova posição candidata e, então, se as condições forem verdadeiras, a candidata é adicionada ao objeto "Posicionda" e a avaliação continua para as linhas ainda não preenchidas.

Exemplo

- Um exemplo da execução do programa é a resolução do problema com o parâmetro <**4**> para indicar a resolução do problema com 4 rainhas em um tabuleiro 4 x 4.
- Para executar o programa o usuário deve digitar a consulta abaixo com o primeiro argumento sendo o N do problema e o segundo qualquer nome de sua preferência para ser o nome da lista de soluções na impressão e acionar o botão Run no canto inferior direito ou por meio atalho **CTRL + Enter**.
- Finalmente o programa gera uma saída com a primeira solução encontrada para a opção de 4 rainhas em um tabuleiro 4 x 4, para as outras soluções do problema o usuário deve precionar o botão Next.

```
?- rainhas(4, Posicao)
```

Compilador para Execução

Abaixo é possível acessar o compilador online utilizado para a execução dos algoritmo

- Prolog - É utilizado o compilador do servidor **swish**, que permite executar o programa no sítio: https://swish.swi-prolog.org

Lisp

Construção do Algoritmo

- O algoritmo foi desenvolvido com a implementação de 3 funções:

  1. <**verifique**>

  – Verifica se uma posição é segura.

  2. <**posicione**>

  – Executa laço para iterar sobre o espaço de busca.

  3. <**imprima**>

  – Imprime soluções encontradas pelo programa.

- O objetivo do programa é posicionar um número **n** de rainhas fornecido pelo usuário.
- O funcionamento do algoritmo consiste em iterar sobre as linhas e colunas do tabuleiro $X_{n,n}$ procurando uma posição segura para cada rainha.
- A verificação de segurança para linhas e colunas é realizada comparando os índices da possível nova posição com os índices de dominância das rainhas já posicionadas.
- A verificação de segurança nas diagonais é confirmada se o valor absoluto da divisão das distâncias entre as rainhas for diferente de 1.
- Caso uma posição selecionada inviabilize o posicionamento de qualquer rainha posterior, o algoritmo retorna e reposiciona as rainhas anteriores de forma recursiva até encontrar uma solução onde as **n** rainhas estarão seguras.

1. Verifique

- A primeira função denominada <**verifique**>, verifica se uma posição em uma determinada casa do tabuleiro é segura.
- As linhas, colunas e diagonais são inspecionadas e a função retorna verdadeiro se as condições de segurança da posição forem satisfeitas.
- As funções <**cond**>, <**member**>, <**mapcar**>, <**car**>, <**cadr**>, <**lambda**> e <**abs**> da biblioteca base de linguagem Common Lisp foram utilizadas.
- A função recebe como parâmetros:

  1. A posição <**x**> no tabuleiro.
  2. A posição <**y**> no tabuleiro.
  3. A lista das posições das rainhas.

- A função de <**cond**> avalia duas condições:

  1. Se a rainha é membra da mesma linha que as rainhas anteriores.

  – A função <**member**> avalia se é verdadeiro o pertencimento da rainha a uma posição segura.
  – A função mapcar avalia cada posição em relação a lista de rainhas.
  – A função lambda recebe com parâmetros a lista xy e aplica o mapcar para linhas e colunas.
  – A expressão lógica or avalia se <**x**> é igual a primeira posição ou <**y**> à segunda.

  2. Se a rainha é membra da mesma diagonal que a das rainhas anteriores.

  – A função <**member**> avalia se é verdadeiro o pertencimento da rainha a uma posição segura.
  – A função mapcar avalia cada posição em relação a lista de rainhas.
  – A função lambda recebe com parâmetros a lista xy e a aplica o mapcar para diagonais.
  – A expressão lógica <**or**> avalia se o valor absoluto da divisão de <**x**> - a primeira posição por <**y**> - segunda posição de <**xy**> é igual a 1.

```
; Define função condição de segurança da rainha na posição x y
(defun verifique (x y rainhas)

  ; Verifica condição de posicionamento da rainha
  (cond

    ; Verifica se rainha da vez é membro da condição de linha
    ((member t (mapcar #'(lambda (xy)
                           (or (= x (car xy)) (= y (cadr xy)))) rainhas)) nil)

    ; Verifica se rainha da vez é membro da condição diagonal
    ((member t (mapcar #'(lambda (xy)
                           (= 1 (abs (/ (- x (car xy)) (- y (cadr xy)))))) rainhas)) nil)

    ; Retorna verdadeiro
    (t)
  )
)
```

2. Posicione

- A segunda função denominada <**posicione**>, itera sobre o tabuleiro.
- A função recebe como parâmetros:

  1. A posição <**x**> no tabuleiro.
  2. A posição <**y**> no tabuleiro.
  3. A lista das posições das <**rainhas**>.
  4. Número máximo <**n**> de rainhas e dimensão do tabuleiro.

- A função de <**cond**> avalia três condições:

  1. Se o tamanho da lista de rainhas posicionadas é igual ao máximo

  – Caso verdadeiro imprime tupla (coluna, linha) da maior para menor.

  2. Se <**x**> ou <**y**> é maior que o máximo

- Caso verdadeiro passa para próxima rainha da lista rainhas.

3. Se <**verifica**> permite o posicionamento.

- Caso verdadeiro adiciona posição a lista rainha caso contrário chama a função verifica de forma recursiva nas posições <**x**> + 1 e <**y**> + 1.

```
; Define posicionamento recursivo da rainha em x e y na ordem: (1 1) ~ (n n)
(defun posicione (x y rainhas n)

  ; Condição de posicionamento seguro
  (cond

    ; Se verdadeiro Imprime tuplas (coluna linha) de posições da solução
    ((= n (length rainhas)) (print (list 'Solução rainhas)) (cdr rainhas))

    ; Caso contrário passa para proxima (coluna linha)
    ((or (> x n) (> y n)) (cdr rainhas))

    ; Verifica se pode posicionar a rainha
    ((verifique x y rainhas)

      ; Define conjunto, aplica laço recursivo com contador x + 1 e empilha rainha
      (setq rec (posicione (+ 1 x) 1 (append (list (list x y)) rainhas) n))

      ; Verifica condição de coluna
      (cond

        ; Condição de laço recursivo com contador y + 1 e conjunto rec verdadeiro
        ((equal rainhas rec) (posicione x (+ 1 y) rainhas n))
        (t rec)
      )
     )

    ; Executa a função de forma recursiva incrementando coluna y
    (t (posicione x (+ 1 y) rainhas n))
  )
)
```

3. Imprima

- A terceira função denominada <**imprima**>, imprime os resultados do algoritmo no formato de tuplas da última coluna para a primeira.
- A função recebe como parâmetro <**n**> que define o número de rainhas e o tamanho do tabuleiro.
- Foram defindas 3 excessões para o parâmetro <**n**>:

  1. Verifica se <**n**> é número.
  2. Verifica se <**n**> é número inteiro.
  3. Verifica se <**n**> é número positivo.

- Executa a função posicione começando com <**x**> = 1 e <**y**> = 1 e o parâmetro <**n**>.

```
; Define função
(defun imprima (n)

    ; Verifica erro de exceção para entrada de caracteres
    (assert (numberp n) (n) "Erro: O parâmetro <n> precisa ser um número.")

    ; Verifica erro de exceção para número real
    (assert (integerp n) (n) "Erro: O parâmetro <n> precisa ser um número inteiro.")

    ; Verifica erro de exceção para entrada de número negativo
    (assert (not (< n 1)) (n) "Erro: O parâmetro <n> precisa ser um número positivo.")

    ; Executa a função posicione
    (posicione 1 1 '() n)
)
```

Exemplo

- Um exemplo da execução do programa é a resolução do problema com o parâmetro <**4**> para indicar a resolução do problema com 4 rainhas em um tabuleiro 4 x 4.

```
; Imprime solução com 4 rainhas
(print (list 'Solução (imprima 4)))
```

```
(SOLUÇÃO ((4 3) (3 1) (2 4) (1 2)))
(SOLUÇÃO ((4 2) (3 4) (2 1) (1 3)))
(SOLUÇÃO NIL)
```

Compilador para Execução

Abaixo é possível acessar o compilador online utilizado para a execução dos algoritmo

- Lisp - É utilizado o compilador do servidor **rextester**, que permite executar o programa no sítio: https://rextester.com

Referências

Java

- https://homepages.dcc.ufmg.br/ bigonha
- www.tutorialspoint.com/java
- You Tube

Prolog

- https://swi-prolog-reference
- https://www.tutorialspoint.com/prolog/
- You Tube

Lisp

- Pratical Common Lisp
- www.tutorialspoint.com/lisp
- You Tube