Ahsanullah University of Science and Technology

Department of Computer Science And Engineering

Course No: 4108

Course Title: Artificial Intelligence Lab

Assignment No: 02

Session: Spring-20

Date of Submission: 31st Jan, 2021

Submitted To:

Tonmoy Hossain
Lecturer, Department Of CSE
Rayhan Ahmed
Lecturer, Department Of CSE

Submitted by:

Name: Faiza Anan Noor
Year: 4th
Semester: 1st
Lab Group: B2
ID: 17.01.04.093

Lab Exercise :

Questions :

1. **1.** Define a recursive procedure in Python and in Prolog to find the sum of 1st n terms of an equal-interval series given the 1st term and the interval.
2. **2.** Define a recursive procedure in Python and in Prolog to find the length of a path between two vertices of a directed weighted graph.
3. **3.** Modify the Python and Prolog codes demonstrated above to find h_2 and h_3 discussed above.

Ans to the Question no. 1 :

Code in Prolog:

```
sum1(1, 100):- !.

sum1(N, S):- N1 is N-1, sum1(N1, S1), S is S1+100+(N-1)*5.
```

Explanation :

In our question we had the following scenario and series:

$$100+105+110+ \dots +(100+(n-1)\times 5)$$

As we can tell,Sum of the 1 st one element is 100.

Sum of the 1 st n elements is the sum of 1 st n-1 elements and the nth element, which is 100+(n-1)x5.

So in short,for our series,

first term : 100

interval : 5.

Let's analyze the important terms in our code.

<u>**sum1(1, 100):-!**</u> This term denotes that if there is only one term which is first term then our sum will be 100 as it is the only term in the series.

<u>**sum1(N, S):-**</u> N Is the number of terms of our series and S is used for storing the sum of the N terms in our series.

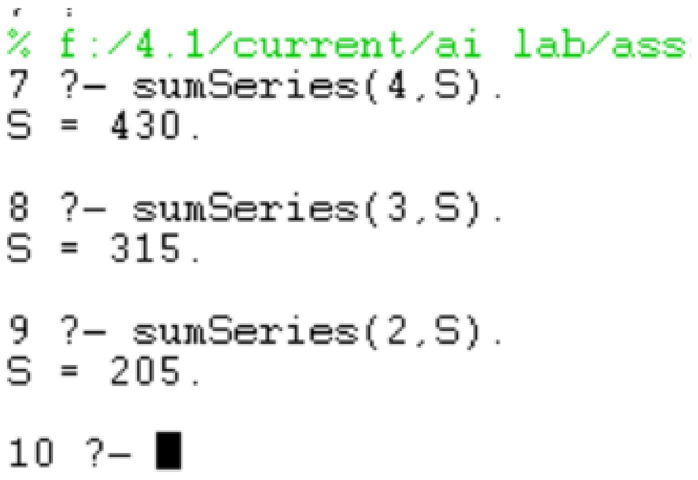
(N-1)*5 is the term that needs to be added to 100 for generating each term as the value of N keeps changing too. So for the 2nd term, N:1,

So out 2nd term will be:

100+(2-1)*5 = 100+5 =**105.**

So basically, N1 is N-1 means if there are 4 terms then we will count from 3 and follow up with recursive procedure for generating our next terms after the first term 100. So every time we generate our new term by adding 100 to the (N-1)*5, we also add it to our S variable for keeping the sum.

Prolog Output :



```
% f:/4.1/current/ai lab/ass
7 ?- sumSeries(4,S).
S = 430.

8 ?- sumSeries(3,S).
S = 315.

9 ?- sumSeries(2,S).
S = 205.

10 ?- █
```

- By calling sumSeries(4,S) we are finding the sum of the first 4 terms which is 430.
- By calling sumSeries(3,S) we are finding the sum of the first 4 terms which is 315.
- By calling sumSeries(2,S) we are finding the sum of the first 4 terms which is 205.

Code In Python :

```
def ssum(N,I,F):
    if (N==0):
        return 0
    elif (N>=1):
        return ssum(N-1,I,F)+F+(N-1)*I
# Main
t=int(input('How many times?'))
for i in range(t):
    print('Iteration:',i+1)
    f=int(input('First element:'))
    d=int(input('Interval:'))
    n=int(input('n:'))
    print('Series sum:', ssum(n,d,f))
```

Explanation :

Here we are using the recursive function ssum to calculate the sum of our series, given our first term and the number of terms and the interval.

In our code,

n : Number of terms .

F : First term

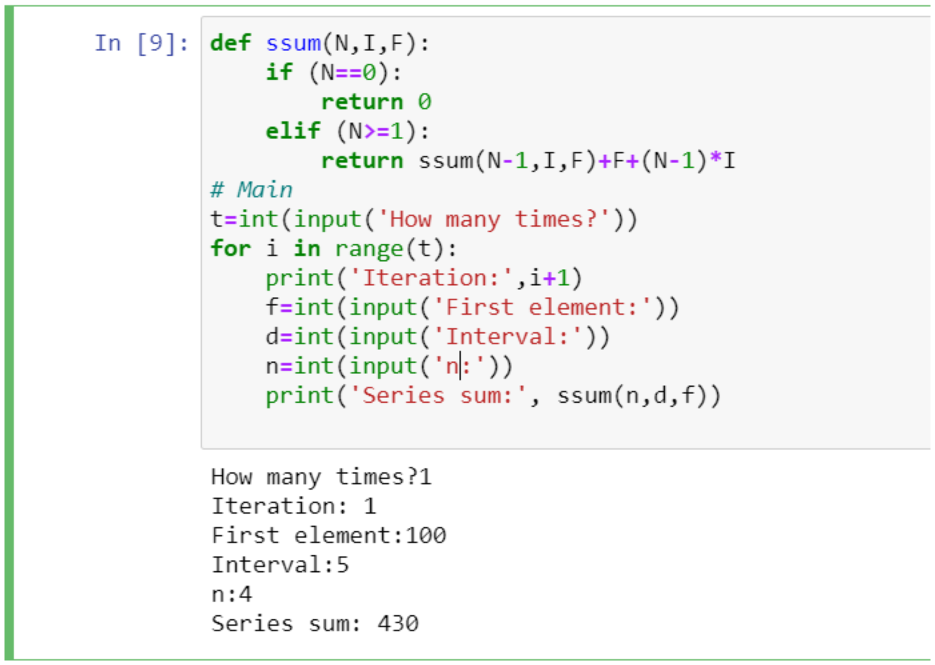
d : Interval.

t: No. of Iterations we want to do.

The first if statement depicts that if there are zero terms then then sum that will be calculated and returned is zero.

Otherwise, ssum function is called recursively and we each time we generate other terms and find out our sum by adding each term generated.

Python Output :



```
In [9]: def ssum(N,I,F):
            if (N==0):
                return 0
            elif (N>=1):
                return ssum(N-1,I,F)+F+(N-1)*I
        # Main
        t=int(input('How many times?'))
        for i in range(t):
            print('Iteration:',i+1)
            f=int(input('First element:'))
            d=int(input('Interval:'))
            n=int(input('n:'))
            print('Series sum:', ssum(n,d,f))
```

```
How many times?1
Iteration: 1
First element:100
Interval:5
n:4
Series sum: 430
```

Here firstly we were asked to input the number of iterations we want to do. Then we are asked to input our First Element, Interval, and number of terms.

Here we wanted just 1 iteration and we gave our input of first element as 100. The Interval was given as 5 and there are only 4 terms and the appropriate output of the series was shown as 430.

Ans to the Q no. 2

Code In Prolog:

```
neighbor(a,c,2). neighbor(a,b,1). neighbor(b,f,4). neighbor(f,h,5).
neighbor(f,g,6).

neighbor(g,h,8). neighbor(c,d,3). neighbor(d,e,7). neighbor(d,f,4).
neighbor(e,g,2).

neighbor(f,g,2).

pathLength(X,Y,L):- neighbor(X,Y,L),!.

pathLength(X,Y,L):- neighbor(X,Z,L1),pathLength(Z,Y,L2), L is L1+L2.
```

Explanation:

neighbor(X,Y,L),!:-

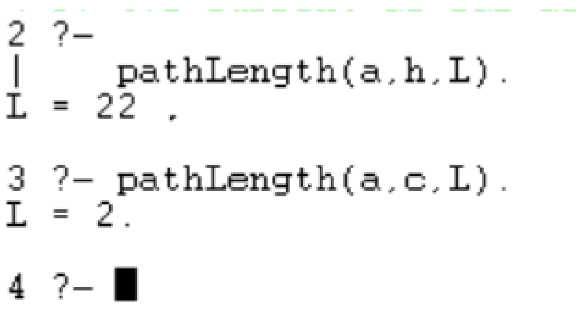
This a predicate containing the beginning and end vertex of an edge and their weight.

pathLength(X,Y,L):- neighbor(X,Y,L),!.

This denotes whether or not the desired given nodes are connected directly. If they are, then the weight will be stored in L as is.

Otherwise, if the nodes are not connected directly then it is divided into two parts and a recursive mechanism is followed, where we keep the weights of each path we traverse in L1, and when we backtrack we add L2 values to them and the final output is in L.

Prolog Output :



```
2 ?-
|    pathLength(a,h,L).
L = 22 ,

3 ?- pathLength(a,c,L).
L = 2.

4 ?- 
```

- Firstly, a and h aren't directly connected so the recursive process is followed until we get the actual weight of the path from a to h which is 22.
- Secondly, a and c are connected directly and the weight of their edge is shown directly.

Python Code :

```
def pathLength(X,Y):


  if X == Y:
    all_paths_weight.append(list(weight))


  for i in range(len(tupleList)):
    if  ((done[i] == 0) & (tupleList[i][0] == X)):
      done[i] = 1
      weight.append(tupleList[i][2])
      pathLength(tupleList[i][1],Y)
  if len(weight) >= 1:
    weight.pop()

tupleList=[('i','a',35), ('i','b',45), ('a','c',22), ('a','d',32),('b','d',28),
       ('b','e',36),('b','f',27),('c','d',31),('c','g',47),('d','g',30),('e','g',26)]

I=int(input('Enter No of Iterations: '))
k=0

while(k<I):
  done= [0 for i in range(len(tupleList))]
  all_paths_weight = []
  weight=[]

  sum2=0
  minSum=99999

  X=str(input('Enter starting vertex: '))
  Y=str(input('Enter ending vertex: '))
  pathLength(X,Y)


 for i in range(len(all_paths_weight)):
    sum2=0
```

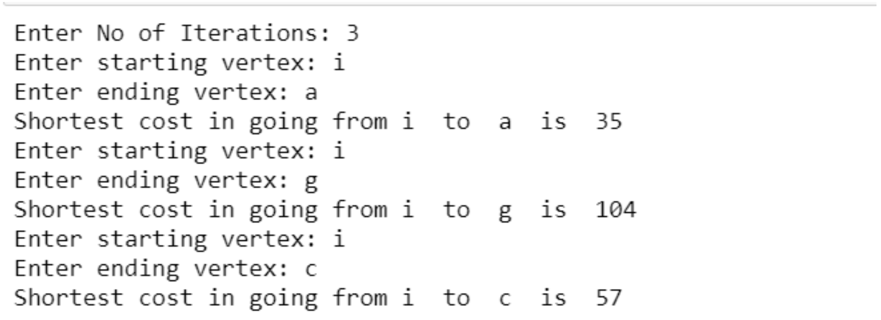
```
        for j in range(len(all_paths_weight[i])):
            sum2+=all_paths_weight[i][j]



        if(sum2<minSum):
            minSum=sum2



    k=k+1
    print('Shortest cost in going from', X,' to ',Y, ' is ',minSum)
```

Explanation:

Here the logic is if starting node is also the ending then there is no cost. pathLength is the recursive function here which will be called by two nodes starting nodes and ending nodes. If starting node is found in [i][0]

Then it will register the weight and its end node if the end node match then the weight will be printed out and if it is not the end node then it will go where then end node is the start node and then if the end node match then the weight is the sum of those two weights.if not then again the function is called.

Python Output:



```
Enter No of Iterations: 3
Enter starting vertex: i
Enter ending vertex: a
Shortest cost in going from i  to  a  is  35
Enter starting vertex: i
Enter ending vertex: g
Shortest cost in going from i  to  g  is  104
Enter starting vertex: i
Enter ending vertex: c
Shortest cost in going from i  to  c  is  57
```

Here, we wanted 3 iterations.

- Firstly we calculated the distance from I to a which is a direct distance which is 35, since they are direct neighbours
- Secondly we calculated the shortest distance from I to g which is 104
- Thirdly we calculated the shortest distance from I to c which is 57

Ans to the Question 3:

Prolog Code for h2:

gtp(1,1,1). gtp(2,1,2). gtp(3,1,3). gtp(4,2,3). gtp(5,3,3). gtp(6,3,2). gtp(7,3,1). gtp(8,2,1). gblnk(2,2). tp(1,1,2). tp(2,1,3). tp(3,2,1). tp(4,2,3). tp(5,3,3). tp(6,2,2). tp(7,3,2). tp(8,1,1). tblnk(3,1).

go:- calcH(1,[],L), write(L),tab(4), sumList(L,V),write('Heuristics 2: '),write(V).

calcH(9,X,X):-!.

calcH(T,X,Y):- dist(T,D), append(X,[D],X1), T1 is T+1, calcH(T1,X1,Y).

dist(T,V):-tp(T,A,B), gtp(T,C,D), V is abs(A-C) + abs(B-D).

sumList([],0):-!. sumList(L,V):-L=[H|T], sumList(T,V1), V is V1+H.

Explanation:

Tp(1,1,2): This refers to the current tile state.

Gtp(1,1,1): This refers to the goal tile state.

Now let’s analyze these terms:

calcH(9,X,X):-!. calcH(T,X,Y):- dist(T,D), append(X,[D],X1), T1 is T+1, calcH(T1,X1,Y). dist(T,V):-tp(T,A,B), gtp(T,C,D), V is abs(A-C) + abs(B-D).

This rule helps us add the shortest Manhattan distance needed to move a tile to the goal state and it appends it in a list to figure out the total no of movements later on.

sumList([],0):-!.

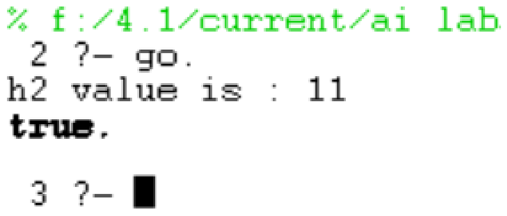
sumList(L,V):-L=[H|T], sumList(T,V1), V is V1+H.:

This rule helps us to calculate Heuristics 2 in recursive way and to find the smallest number of movements needed to move the current state to goal state.

go:- calcH(1,[],L), write(L),tab(4), sumList(L,V),write('Heuristics 2: '),write(V). :

This prints out the total number of movements on the console.

Prolog Output:



```
% f:/4.1/current/ai lab
 2 ?- go.
h2 value is : 11
true.

 3 ?- 
```

Here if we run go, then we'll get the total number of movements needed to bring the current state to goal state by using the formula of Manhattan distance for each of the tiles and by keeping them in a List first.

Code in Python for h2:

```
gtp=[(1,1,1),(2,1,2),(3,1,3),(4,2,3),(5,3,3),(6,3,2),(7,3,1),(8,2,1)]

gblnk = (2,1)


tp=[(1,1,2),(2,1,3),(3,2,1), (4,2,3), (5,3,3), (6,2,2), (7,3,2),(8,1,1)]

blnk = (3,1)



i,H=0,0

L = []

LEN= len(gtp)

for i in range(LEN):

   val = abs(gtp[i][1] -tp[i][1] ) + abs(gtp[i][2] - tp[i][2])
```

```
    H += val

    L.append(val)



print('T: ' , L)

print('H value',H)
```

Explanation:

Let’s analyze the important terms:

Tp(1,1,2): This belongs to a tuple which contains the current tile state.

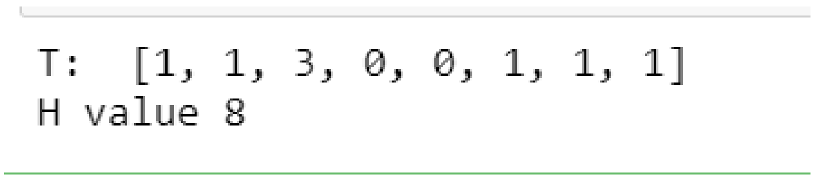
Gtp(1,1,1): This belongs to a tuple which contains the goal tile state.

Now let’s explain the term below:

```
for i in range(LEN):

    val = abs(gtp[i][1] -tp[i][1] ) + abs(gtp[i][2] - tp[i][2])

    H += val

    L.append(val)
```

Here we are calculating the Manhattan distance for moving each of the tiles from current tile state to goal state and we are incrementing H every time. We are also appending the number of moves needed for each of the tile in a List.

OUTPUT:



```
T:  [1, 1, 3, 0, 0, 1, 1, 1]
H value 8
```

Here we calculated the total number of movements needed to bring the current tile state to the goal state.

All the shortest movements are kept in a tile and then the H value was measured to be 8.

Code In Prolog for h3:

```
:-dynamic(hval/1).

evalState(L,V):- assert(hval(0)),hl(1,L),
di_up(1,L),di_dn(1,L),hval(V),retractall(hval(_)).

hl(8,_):-!.
hl(I,L):- nthel(I,L,X), chk_incr(I,L,X), I1 is I+1, hl(I1,L).

chk_incr(8,_,_):-!.
chk_incr(I,L,X):- I1 is I+1, nthel(I1,L,Y),do_incr(X,Y),chk_incr(I1,L,X).

do_incr(X,Y):- X=Y, incr_hval.
do_incr(_,_).

incr_hval:-hval(V), V1 is V+1, retract(hval(_)), assert(hval(V1)).

di_up(8,_):-!.
di_up(I,L):- nthel(I,L,X), chkup_incr(I,L,X,0), I1 is I+1,di_up(I1,L).

chkup_incr(8,_,_,_):-!.
chkup_incr(I,L,X,K):- I1 is I+1, nthel(I1,L,Y), K1 is K+1, doup_incr(X,Y,K1),
chkup_incr(I1,L,X,K1).

doup_incr(X,Y,K1):- X1 is X+K1, Y=X1, incr_hval.
doup_incr(_,_,_).

di_dn(8,_):-!.
di_dn(I,L):- nthel(I,L,X), chkdn_incr(I,L,X,0), I1 is I+1,di_dn(I1,L).

chkdn_incr(8,_,_,_):-!.
chkdn_incr(I,L,X,K):- I1 is I+1, nthel(I1,L,Y), K1 is K+1,dodn_incr(X,Y,K1),
chkdn_incr(I1,L,X,K1).

dodn_incr(X,Y,K1):- X1 is X-K1, Y=X1, incr_hval.
dodn_incr(_,_,_).
```

```
nthel(N,[_|T],El):- N1 is N-1, nthel(N1,T,El).

nthel(1,[H|_],H):-!.


findNum :- evalState([6, 1, 5, 7, 4, 3, 8, 1],V), write('No of attacking pairs : '),
write(V), tab(5), fail.

findNum.
```

Explanation:

Let’s analyze and evaluate the most important terms:

evalState([6, 1, 5, 7, 4, 3, 8, 1],V) : The first parameter is the position of all the 8 queens

evalState(L,V): To count and check the attacking pairs of queens and and put the value in V in recursive way.

h1(): to check if the two pairs are in attacking mode in diagonally upward position

di_up(): to check if the two pairs are in attacking mode in diagonally upward position

di_dn(): to check if the two pairs are in attacking mode in diagonally downward position

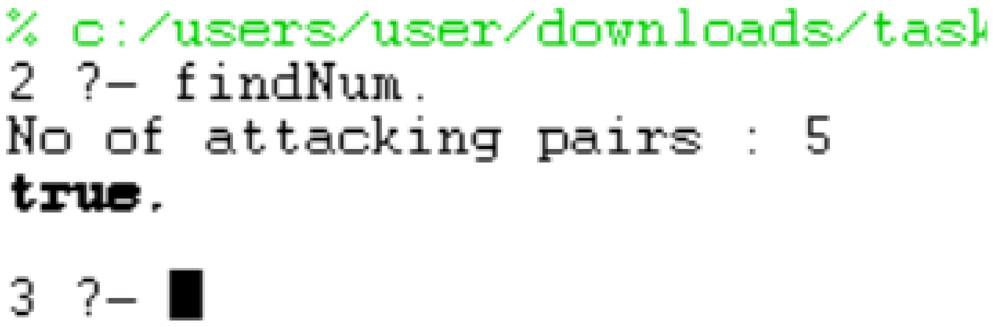
nthel(): to get the nth element of a list. It gets two possible attacking pairs each time.

Incr_val(): to increase the value of attacking pair count.

Assert() and retract() are used to update/store value of hval and to remove the value respectively. retractAll() is used to discard all the values.

findNum: This is used to start the program to find out the number of attacking pairs.

Prolog Output:



```
% c:/users/user/downloads/task
2 ?- findNum.
No of attacking pairs : 5
true.

3 ?- 
```

Here we called findNum method to calculate the total Number of attacking pairs for our given board. In our case the No of attacking pairs was found to be 5.

Code In Python for h3:

```
matrix = [(0,0,0,0,0,0,'Q',0),
          (0,0,0,'Q',0,0,0,0),
          ('Q',0,0,0,0,0,0,0),
          (0,0,'Q',0,0,0,0,0),
          (0,0,0,0,'Q',0,0,0),
          (0,0,0,0,0,'Q',0,0),
          (0,0,0,0,0,0,0,0),
          (0,'Q',0,0,0,0,0,'Q')]


L = []
```

```
ROW=len(matrix)

COL=len(matrix[0])

for i in range(ROW):

        for j in range(COL):

                if matrix[i][j] == 'Q':

                        L.append([i, j])


forward_attack = 0

for k in range(len(L)):

   row =  L[k][0]

   col = L[k][1]

   for i in range(col+1, COL):

      if matrix[L[k][0]][i] == 'Q' :

         forward_attack =forward_attack+ 1



upward_attack = 0


for k in range(len(L)):

   j = 1

   for i in reversed(range(0,L[k][0]-1)):

      newCol = L[k][1]+j
```

```
        if i <0 or newCol == COL:
            break
        if matrix[i][newCol] == 'Q' :
            upward_attack =upward_attack+ 1
        j =j + 1


downward_attack = 0
for k in range(len(L)):
    col = L[k][1]
    j = 1
    for i in range(L[k][0]+1,ROW):
        newRow=L[k][0]+j
        newCol=L[k][1]+j
        if newCol == COL:
            break
        if matrix[newRow][newCol] == 'Q' :
            downward_attack = downward_attack +1
        j =j+ 1


print('(h3): ','forward_attack + downward_attack + upward_attack: ', 
forward_attack,' + ',downward_attack , ' + ', upward_attack, ' = ',  forward_attack 
+ downward_attack + upward_attack)
```

Explanation:

For this code, we used 3 different ways to calculate the number of attacking pairs for 3 different cases. We kept the Row and column positions of our queens in a list called L.

Face to Face:

For calculating face to face attacking pairs, the code snippet is:

```
forward_attack = 0

for k in range(len(L)):

   row =  L[k][0]

   col = L[k][1]

   for i in range(col+1, COL):

      if matrix[L[k][0]][i] == 'Q' :

         forward_attack =forward_attack+ 1
```

Here we traverse the columns for one particular row, and whenever we find one queen face to face we increment the value of attacking pair count by 1 for the case of measuring face to face attacking pairs.

Diagonally Upwards

```
upward_attack = 0


for k in range(len(L)):

   j = 1

   for i in reversed(range(0,L[k][0]-1)):
        newCol = L[k][1]+j
        if i <0 or newCol == COL:
           break
        if matrix[i][newCol] == 'Q' :
           upward_attack =upward_attack+ 1
        j =j + 1
```

Here we traverse diagonally upward by incrementing the column number of our current position and by incrementing the row of our current position. When we do so, if we encounter a ‘Q’ in our board or if we meet another queen, then we incrementing the value of attacking pair count in downward position by 1 each time.

Diagonally Downwards:

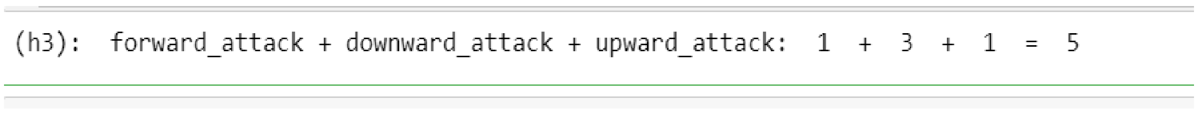
For calculating the number of queens who are in attacking mode in diagonally downward position we use the following code:

```
downward_attack = 0
for k in range(len(L)):
   col = L[k][1]
   j = 1
   for i in range(L[k][0]+1,ROW):
      newRow=L[k][0]+j
      newCol=L[k][1]+j
      if newCol == COL:
        break
    if matrix[newRow][newCol] == 'Q' :
        downward_attack = downward_attack +1
    j =j+ 1
```

Here we traverse diagonally downward by incrementing the column number of our current position and by decrementing the row of our current position. When we do so, if we encounter a ‘Q’ in our board or if we meet another queen, then we incrementing the value of attacking pair count in downward position by 1 each time.

Finally, we add up all the 3 variables to generate h3.

Python Code Output:



```
(h3):  forward_attack + downward_attack + upward_attack:  1  +  3  +  1  =  5
```

For this case, we calculated the number of attacking pairs for our queens.

In our case,

Upward attacking pair No: 1

Face to Face attacking pair No: 1

Downward attacking pair No: 3

So the value of h3: 1+1+3 = **5**