Requirement a

- Input: array with `8` element which represent for queens.The queen can be placed or not.The queens which are placed must not attack each together. If element `-1` is queen still isn't placed and element `from 1 to 64` is at which place the queen is placed
- Ouput: array with `8` element. Element `-1` doesn't exist in this array and entire queens doesn't attack each together
- For simplier calculation, i will convert element `from 1 to 64` to 2 dimension array `(x,y)` most of time in the program. In this form. we can consider whether the queen attack other queens or not.
  For example:
  - ☑ element `9` will convert to `(0,1)`
  - ☑ element `12` will convert to `(3,1)`
  - ☑ queen at `(0,1)` attack queen at `(3,1)` because they are in the same row `1`
- Initial state can be `[-1,-1,-1,-1,-1,-1,-1,-1]` or `[4,10,-1,-1,-1,-1,-1,-1]`
- Goal state will look like `[4, 10, 23, 27, 38, 48, 53, 57]`. If we convert to form `(x,y)` and represent queens by character `Q` in 2d array, it will like this:

```
. . . Q . . . .

. Q . . . . . .

. . . . . . Q .

. . Q . . . . .

. . . . . Q . .

. . . . . . . Q

. . . . Q . . .

Q . . . . . . .
```

Requirement e

- We represent state in the program by `class State`

> One state will have unique `id` to find from the frontier, `board` to carry the input that i have mentioned above `father` is the state that generate this state, `hvalue` is heuristic value and `gvalue` is path cost

```
class State:
    def __init__(self,board: List[int],father:State = None):
        self.board = board
```

```
        self.id = uuid4()

        self.hvalue = self.getHeuristicValue()
        self.father = father

        if father:
            self.gvalue = father.gvalue + 1
        else:
            self.gvalue = 0
```

> My heuristic function will count the amount of the pairs of queens that attack each together (attack on row, column or diagonal)

```
class State:
    def __isSameDiagonal(self,firstQueen:Position,secondQueen:Position):
        boardSize = len(self.board)
        forwardPoint = deepcopy(firstQueen)
        while forwardPoint.validate(boardSize):
            if forwardPoint == secondQueen:
                return True
            forwardPoint.forwardDiagonalIncrease()
        forwardPoint = deepcopy(firstQueen)
        while forwardPoint.validate(boardSize):
            if forwardPoint == secondQueen:
                return True
            forwardPoint.forwardDiagonalDecrease()

        backwardPoint = deepcopy(firstQueen)
        while backwardPoint.validate(boardSize):
            if backwardPoint == secondQueen:
                return True
            backwardPoint.backDiagonalIncrease()
        backwardPoint = deepcopy(firstQueen)
        while backwardPoint.validate(boardSize):
            if backwardPoint == secondQueen:
                return True
            backwardPoint.backDiagonalDecrease()

        return False


    def getHeuristicValue(self):
        attackPairs = 0
        boardSize = len(self.board)
        for i in range(boardSize):
            for j in range(i+1,boardSize):
                if self.board[i] == -1 or self.board[j] == -1:
                    continue
                firstQueen = Position.getPosIn2DArray(self.board[i],boardSize)
                secondQueen = Position.getPosIn2DArray(self.board[j],boardSize)
                isSameRow = firstQueen.y == secondQueen.y
```

```
                isSameColumn = firstQueen.x == secondQueen.x
                isSameDiagonal = self.__isSameDiagonal(firstQueen,secondQueen)
                if isSameRow or isSameColumn or isSameDiagonal:
                    attackPairs += 1

        return attackPairs
```

> To acquire successors, the state firstly find whether entire queens are placed or not. If all of queens was placed, i will move queen around the board which not queen at that position If still queens wasn't placed, i will place queen with the same way i move the queen.

```
class State:
    def action(self,currentQueen:int,value:int):
        board = deepcopy(self.board)
        board[currentQueen] = value
        successor = State(board,self)
        return successor

    def generateSuccessors(self,initPos,currentPos):
        successors:List[State] = []
        boardSize = len(self.board)

        #move queen when entire queens placed
        if not -1 in self.board:
            for j in range(boardSize*boardSize):
                if j+1 in self.board:
                    continue
                successor = self.action(currentPos,j+1)
                successors.append(successor)

            currentPos += 1
            if currentPos == boardSize:
                currentPos = initPos

        #any queens haven't placed in the board yet
        else:
            for i in range(boardSize):
                if self.board[i] == -1:
                    for j in range(boardSize*boardSize):
                        if j+1 in self.board:
                            continue
                        successor = self.action(i,j+1)
                        successors.append(successor)

        return successors
```