

## CP468 Assignment 2

Francis, Alexander  
fran0950@mylaurier.ca

Laslavic, Nicole  
lasl7800@mylaurier.ca

Iskander, Keven  
iska4540@mylaurier.ca

Castaneda, Carla  
cast4730@mylaurier.ca

November 9, 2020

## Contents

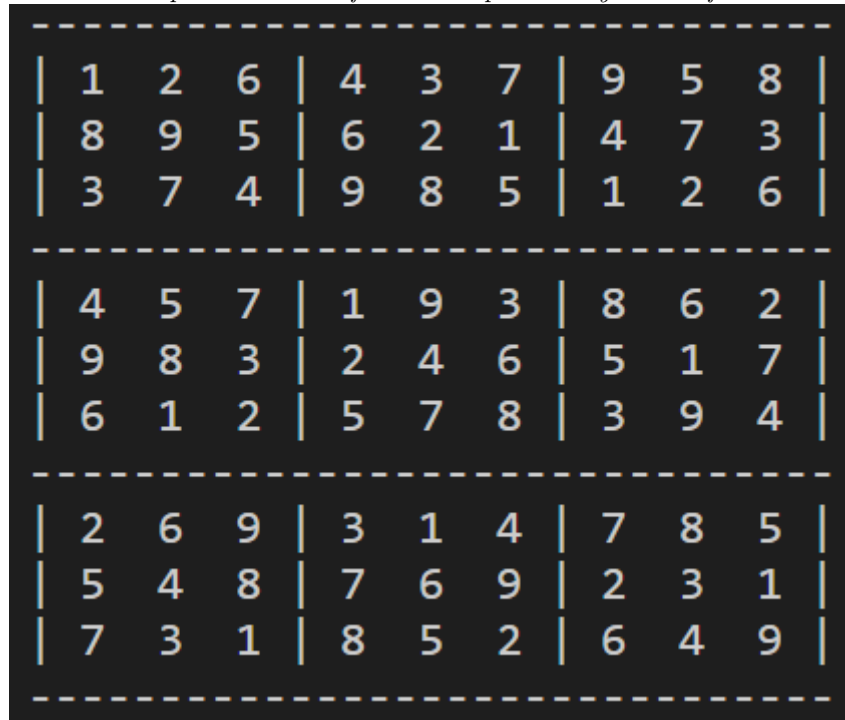
1	Introduction	2
2	Sudoku Rules	2
3	Algorithm Implementation	3
3.1	AC-3 . . . . .	3
3.2	Backtracking . . . . .	3
4	Input and Output Examples	3
4.1	<b>Easy Puzzle</b> . . . . .	4
4.2	<b>Medium Puzzle</b> . . . . .	5
4.3	<b>Hard Puzzle</b> . . . . .	6
5	Puzzle Encoding	7
6	Final Notes	7
7	Source Code	8
7.1	sudoku.py . . . . .	8
7.2	utilities.py . . . . .	16

## 1 Introduction

**GitHub Repository:** [SudokuAC3](#)

Tasked with the problem of solving a 9x9 Sudoku puzzle, our implementation will use the constraint-satisfaction AC3 algorithm to fill in the missing components of the puzzle.

*The completed version of a Sudoku puzzle may look as follows:*



1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

## 2 Sudoku Rules

The rules of completing a Sudoku puzzle are fairly simple to understand. Firstly, each possible input has the following domain, where  $x \in 1, 2, 3, 4, 5, 6, 7, 8, 9$ . When deciding an appropriate value for a given input position, there are three main rules to consider:

1. No row may have any duplicate values
2. No column may have any duplicate values
3. No sub-square may contain any duplicate values. The 9x9 puzzle consists of nine 3x3 sub-squares, also arranged in a 3x3 fashion. In the image above, each sub-square is separated by hyphens and line breaks.

## 3 Algorithm Implementation

### 3.1 AC-3

To implement the AC-3 algorithm, every alldiff constraint was organised into a tuple. The first element of the tuple contains the current node, and the second element contains the neighbour constraint, which is any node in the same row, column, and sub-square. The binary constraints (tuples) managed by a queue and removed to be processed in the revise function. The domain of the first node in the tuple will potentially have an item in its domain removed if there is no value in the domain of the second node that satisfies the constraint between the two. If the domain was revised, all neighbours of that node will be inserted into the queue. This time the neighbour node will be the first item in the tuple so that the domain of the neighbour is revised according to the current node. If at any point the domain of a node is empty, there is an inconsistency within the Sudoku puzzle that cannot be solved using the AC-3 algorithm alone and the function will return false.

### 3.2 Backtracking

The backtracking algorithm works by iterating through the empty squares in the matrix and inserting nodes from a list of possible values. While iterating through, if all possible values are exhausted for a single square the algorithm will iterate backwards to the previously looked at square to try another value. This function uses the helper functions 'is.valid()' and 'MRV\_heuristic()'. The is.valid() function checks each node to make sure there are no repeat values in the row, column and sub-squares. The MRV\_heuristic() function finds the next empty square with the shortest length of possible values to increase efficiency.

## 4 Input and Output Examples

The following input and output examples demonstrate how our implementation of the AC3 and backtracking algorithms work on a given puzzle. The puzzle with the "BEFORE" label is the initial puzzle. Each input puzzle has been taken from the following source: [Example Puzzles and Solutions](#)

## 4.1 Easy Puzzle

BEFORE:

					2	6			7		1	
	6	8				7				9		
	1	9					4		5			
<hr/>												
	8	2			1					4		
			4		6		2		9			
		5					3			2	8	
<hr/>												
			9		3					7	4	
		4				5				3	6	
	7		3			1	8					

AFTER AC3:

	4	3	5		2	6	9		7	8	1	
	6	8	2		5	7	1		4	9	3	
	1	9	7		8	3	4		5	6	2	
<hr/>												
	8	2	6		1	9	5		3	4	7	
	3	7	4		6	8	2		9	1	5	
	9	5	1		7	4	3		6	2	8	
<hr/>												
	5	1	9		3	2	6		8	7	4	
	2	4	8		9	5	7		1	3	6	
	7	6	3		4	1	8		2	5	9	

AFTER BACKTRACKING:

	4	3	5		2	6	9		7	8	1	
	6	8	2		5	7	1		4	9	3	
	1	9	7		8	3	4		5	6	2	
<hr/>												
	8	2	6		1	9	5		3	4	7	
	3	7	4		6	8	2		9	1	5	
	9	5	1		7	4	3		6	2	8	
<hr/>												
	5	1	9		3	2	6		8	7	4	
	2	4	8		9	5	7		1	3	6	
	7	6	3		4	1	8		2	5	9	

Total Execution Time: 2.651 seconds

## 4.2 Medium Puzzle

BEFORE:

			3			7				
	2	7				6	9			
	9					2	3		7	8
	5	8	4			7			3	6
		6	7			8			5	4
										9
	6		8		7	3				4
					4	9			2	5
					2				6	

AFTER AC3:

	8	1	3		5	4	7		2	9	6	
	2	7	5		8	6	9		4	3	1	
	9	4	6		1	2	3		7	5	8	
	5	8	4		9	7	1		3	6	2	
	3	9	2		6	5	4		1	8	7	
	1	6	7		3	8	2		5	4	9	
	6	2	8		7	3	5		9	1	4	
	7	3	1		4	9	6		8	2	5	
	4	5	9		2	1	8		6	7	3	

AFTER BACKTRACKING:

	8	1	3		5	4	7		2	9	6	
	2	7	5		8	6	9		4	3	1	
	9	4	6		1	2	3		7	5	8	
	5	8	4		9	7	1		3	6	2	
	3	9	2		6	5	4		1	8	7	
	1	6	7		3	8	2		5	4	9	
	6	2	8		7	3	5		9	1	4	
	7	3	1		4	9	6		8	2	5	
	4	5	9		2	1	8		6	7	3	

Total Execution Time: 3.437 seconds

### 4.3 Hard Puzzle

BEFORE:

		2								3	
		7	4		6		8				
							3			2	
	6	8			5	4			1		
	5					1			7	8	
						9					
									4		

AFTER AC3:

		2								3	
		7	4		6		8				
							3			2	
	6	8			5	4			1		
	5					1			7	8	
						9					
									4		

AFTER BACKTRACKING:

	1	2	6		4	3	7		9	5	8	
	8	9	5		6	2	1		4	7	3	
	3	7	4		9	8	5		1	2	6	
	4	5	7		1	9	3		8	6	2	
	9	8	3		2	4	6		5	1	7	
	6	1	2		5	7	8		3	9	4	
	2	6	9		3	1	4		7	8	5	
	5	4	8		7	6	9		2	3	1	
	7	3	1		8	5	2		6	4	9	

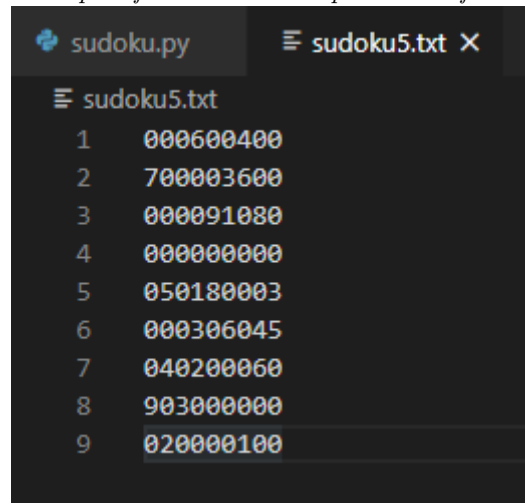
Total Execution Time: 30.807 seconds

## 5 Puzzle Encoding

The puzzle that are provided to the AC3 and backtracking algorithms are prepared as text files with the following constraints:

1. All puzzle values must be integers within the text file
2. No puzzle may exceed 81 digits
3. The file must have 9 digits per row, and 9 rows in total
4. Empty spaces are to be represented as 0.

*An example of a valid Sudoku puzzle text file input:*



```
sudoku.py  sudoku5.txt X
sudoku5.txt
1  000600400
2  700003600
3  000091080
4  000000000
5  050180003
6  000306045
7  040200060
8  903000000
9  020000100
```

Once read, the sudoku text file is converted and initialized as a Puzzle object with a 2D table attribute. Each index of the 2D Sudoku table is filled with a Node. The Node stores and tracks data such as value, neighbors and domain for each tile in the Sudoku Puzzle.

## 6 Final Notes

It was interesting to see how the implementation of the algorithm outlined in this assignment quickly solved some of the puzzles, but took much longer on puzzles that seemed similar in difficulty at first glance. The total computation time after adding the minimum remaining value heuristic significantly decreased, the algorithm performed more than ten times slower without it. Seeing as the goal of the AC3 algorithm is to reduce the domain of each node by making the arc consistent, there are certain puzzles that cannot be solved using AC3 alone. Every time we remove an arc, we crosscheck its consistency and continuously reduce variable domains. Once reduced, we are occasionally met with a valid solution, that is immediately solvable in a way that does not violate the Sudoku puzzle's constraints. However, if an immediate solution is not revealed, backtracking must be executed on the puzzle. Since the domains of each node have now been trimmed by the AC3 algorithm, the process of backtracking will be much more efficient leading us to believe that the optimal solving technique involves a combination of backtracking and AC3.



## 7 Source Code

### 7.1 sudoku.py

```

"""
-----
sudoku.py
9x9 Sudoku Puzzle implementation as described in class
-----
CP468
Assignment 2
Authors: Keven Iskander, Carla Castaneda, Nicole Laslavic, Alexander Francis
__updated__ = "2020-11-09"
-----
"""

import utilities
import time
# Some Sudoku puzzle challenges taken from here:
# https://dingo.sbs.arizona.edu/~sandiway/sudoku/examples.html

class Node:
    domain = [1,2,3,4,5,6,7,8,9]
    value = 0
    def __init__(self, value, domain = domain,row=0,col=0):
        self.value = value
        self.domain = domain
        self.neighbours=[]
        self.row=row
        self.col=col
        return

    def __int__(self):
        return int(self.value)

class Sudoku:
    def __init__(self):
        """
        -----
        Populates sodoku puzzle
        Parameters: self - Matrix
        Return: None
        -----
        """
        self.lvalues = []
        f = open('sudoku8.txt', 'r')
        lines = f.readlines()
        if len(lines)!=9:
            print('ERROR: Invalid puzzle file')
            self.table = [[Node(0) for i in range(9)] for j in range(9)]
        else:
            self.table = [[0 for i in range(9)] for j in range(9)]
            for i in range(len(self.table)):

```

```

        for j in range(len(self.table)):
            self.lvalues.append(int(lines[i][j]))
            self.table[i][j] = Node(int(lines[i][j]))
            self.table[i][j].row=i
            self.table[i][j].col=j
            if self.table[i][j].value !=0:
                self.table[i][j].domain = [self.table[i][j].value]
        for k in range(len(self.table)):
            for l in range(len(self.table)):
                if self.table[k][l].value == 0: self.table[k][l].domain = self.update_domain(k,l)
f.close()
return

def print_table(self):
    """
    -----
    Prints sudoku puzzle
    Parameters: self - Matrix
    -----
    """
    print("-"*31)
    i = 0
    for x in self.table:
        for y in x:
            if i % 9 == 0:
                print("|", end="")
            if y.value == 0:
                print("{} ".format(" "), end="")
            else:
                print("{} ".format(y.value), end="")
            i += 1
            if i % 3 == 0:
                print('|', end="")
            if i % 9 == 0 and i != 0:
                print()
            if i % 27 == 0:
                print("-"*31)
    return

def print_domain(self, row, col):
    """
    -----
    Prints domain of Node at specified index
    Parameters: self - Matrix
                row - row index
                col - column index
    Return: None
    -----
    """
    print(self.table[row][col].domain)
    return

```

```

def valid_col(self, col):
    """
    -----
    Returns if a column is valid.
    Parameters: self - Matrix
                col - column index
    Return: Boolean - True if no repeat numbers (1-9) exist in the column
                False if repeat numbers in column
    -----
    """
    visited = []
    result = True
    for i in range(9):
        if (not self.table[i][col].value in visited):
            if (self.table[i][col].value != 0):
                visited.append(self.table[i][col].value)
        else:
            return result == False
    return result

def valid_row(self, row):
    """
    -----
    Returns if a row is valid.
    Parameters: self - Matrix
                row - row index
    Return: Boolean - True if no repeat numbers (1-9) exist in the row
                False if repeat numbers in row
    -----
    """
    visited = []
    result = True
    for i in range(9):
        if (not self.table[row][i].value in visited):
            if (self.table[row][i].value != 0):
                visited.append(self.table[row][i].value)
        else:
            return result == False
    return result

def valid_subsquare(self, row, col):
    """
    -----
    Returns if a subsquare is valid.
    Parameters: self - Matrix
                col - column index
                row - row index
    Return: Boolean - True if no repeat numbers (1-9) exist in the subsquare
                False if repeat numbers in subsquare
    -----
    """

```

```

-----
"""
visited = []
result = True
r_index = row
c_index = col
#Find top left index of subsquare
r = False
c = False
while r == False or c == False:
    if(r == False):
        if(r_index%3 == 0):
            r = True
        else:
            r_index -= 1
    if(c == False):
        if(c_index%3 == 0):
            c = True
        else:
            c_index -= 1
for row in range(r_index, r_index+3):
    for col in range(c_index, c_index+3):
        if(not self.table[row][col].value in visited):
            if (self.table[row][col].value != 0):
                visited.append(self.table[row][col].value)
        else:
            return result == False
return result

def is_valid(self):
    """
    -----
    Returns if a Sudoku puzzle is valid. Checks each node to see if row, col
    and subsquares are valid.
    Parameters: self - Matrix
                col - column index
                row - row index
    Return: Boolean - True if valid
                False if not valid
    -----
    """
    for i in range(9):
        for j in range(9):
            if self.valid_row(i) == False or self.valid_col(j) == False or self.valid_subsquare(i, j) == False:
                return False
    return True

def update_domain(self, row, col):
    """
    -----
    Returns updated domain for node object where node value does not equal 0.
    -----

```

```

Parameters: self - Matrix
row - row index
col - column index
Return: List - New domain
-----
"""
dom = [1,2,3,4,5,6,7,8,9]
visited = []
if self.table[row][col].value == 0:
    for i in range(9):
        if not self.table[i][col].value == 0 and not self.table[i][col] in visited:
            visited.append(self.table[i][col].value)
    for i in range(9):
        if not self.table[row][i].value == 0 and not self.table[row][i] in visited:
            visited.append(self.table[row][i].value)
    r_index = row
    c_index = col
    r = False
    c = False
    while r == False or c == False:
        if(r == False):
            if(r_index%3 == 0):
                r = True
            else:
                r_index -= 1
        if(c == False):
            if(c_index%3 == 0):
                c = True
            else:
                c_index -= 1
    for row in range(r_index, r_index+3):
        for col in range(c_index, c_index+3):
            if(not self.table[row][col].value in visited):
                if (self.table[row][col].value != 0):
                    visited.append(self.table[row][col].value)
    new_visited = []
    for i in dom:
        if i not in visited:
            new_visited.append(i)
    visited = new_visited
    return visited
return [self.table[row][col].value]

def backtracking(self):
    """
    -----
    Resursively solves sudoku puzzles using backtracking
    Parameters: self - Matrix
    Return: Boolean
    -----
    """
    index = self.MRV_heuristic()

```

```

    if index == (-1, -1):
        return True
    else:
        row = index[0]
        col = index[1]
        self.table[row][col].domain = self.update_domain(row, col)
        for i in self.table[row][col].domain:
            if self.is_valid() == True:
                self.table[row][col].value = i
                if self.backtracking() == True:
                    return True
                self.table[row][col].value = 0
        return False

def MRV_heuristic(self):
    """
    -----
    Finds the minimum remaining value.
    Parameters: self - Matrix
    Return: index - (row, col)
    -----
    """
    temp_min = [1,2,3,4,5,6,7,8,9]
    index = (-1,-1)
    for i in range(9):
        for j in range(9):
            if len(self.table[i][j].domain) < len(temp_min) and self.table[i][j].value == 0:
                temp_min = self.table[i][j].domain
                index = (i,j)
    return index

def find_neighbours(self,i,j):
    """
    -----
    Finds the nodes which constraints of current node called neighbours in this case
    Parameters: self - Sudoku
                i- row
                j-column
    Return: neighbours- list of nodes which are all "neighbours" or constraints of the node in position
    -----
    """
    neighbours=[]
    original_i=i
    original_j=j
    for k in range(len(self.table)):
        if (not self.table[i][k] in neighbours and self.table[i][k]!=self.table[original_i][original_j]):
            neighbours.append(self.table[i][k])
    for l in range(len(self.table)):
        if (not self.table[l][j] in neighbours and self.table[l][j]!=self.table[original_i][original_j]):
            neighbours.append(self.table[l][j])
    r = False

```

```

c = False
while r == False or c == False:
    if(r == False):
        if(i%3 == 0):
            r = True
        else:
            i -= 1
    if(c == False):
        if(j%3 == 0):
            c = True
        else:
            j -= 1
for row in range(i, i+3):
    for col in range(j, j+3):
        if(not self.table[row][col] in neighbours and self.table[row][col] != self.table[original_i][original_j]):
            neighbours.append(self.table[row][col])
self.table[original_i][original_j].neighbours=neighbours
return neighbours

def constraints(self):
    """
    -----
    Gets constraints and arranges in arc pairs
    Parameters: self - Sudoku
    Return: constraints - list of tuples with a constraint pairs
    -----
    """
    constraints=[]
    for i in range (len(self.table)):
        for j in range (len(self.table)):
            neighbours=self.find_neighbours(i,j)
            for k in range (len(neighbours)):
                constraints.append((self.table[i][j],self.table[i][j].neighbours[k]))
    return constraints

def AC3(self,constraints):
    """
    -----
    Arc consistancy algorithm which makes every variable arc-consistent
    Parameters: self - Sudoku
               constraints- constraints in pairs (node,neighbour)
    Return: Boolean - True if no inconsistencies found and False otherwise
    -----
    """
    cons_q=utilities.Queue()
    for i in constraints:
        cons_q.insert(i)
    print("length of queue: ",len(cons_q))
    while (cons_q.is_empty()==False):
        arc=cons_q.remove()
        for i in range(len(self.table)):

```

```

        for k in range(len(self.table)):
            if (self.table[i][k].row==arc[0].row and self.table[i][k].col==arc[0].col):
                node=self.table[i][k]
        revised=self.revise(node,arc[1])
        if (revised[0]):
            if (len(revised[1].domain)==0):
                return False
            for neighbour in revised[1].neighbours:
                if (neighbour!=arc[1]):
                    for j in range(len(neighbour.neighbours)):
                        if (neighbour.neighbours[j]==node):
                            neighbour.neighbours[j]=revised[1]
                    cons_q.insert((neighbour,revised[1]))
        print("length of queue: ",len(cons_q))
    return True

def revise(self,x,y):
    """
    -----
    Checks if theres a a value in x.domain that does not satisfy
    the constraint between c and y
    Parameters: self - Sudoku
                x- node which will be revising domain
                y- neighbour
    Return: revised - Boolean if domain of x was changed will return true else false
    -----
    """
    revised=False
    return_node=x
    for i in x.domain:
        not_satisfied=True
        for j in y.domain:
            if i!=j:
                not_satisfied=False
        if not_satisfied:
            x.domain.remove(i)
            self.table[x.row][x.col]=x
            revised=True
            return_node=self.table[x.row][x.col]
    return revised, return_node

def AC3_table(self):
    """
    -----
    Populates table by using updated domains per Node
    Parameters: self - Sudoku
    Return: revised - None
    -----
    """
    for i in range(9):
        for j in range(9):

```



```

        if (len(self.table[i][j].domain) == 1):
            value = self.table[i][j].domain[0]
            self.table[i][j].value = value

    return

def main():
    start = time.time()
    sud = Sudoku()

    print("BEFORE: ")
    sud.print_table()
    print()

    print("AFTER AC3: ")
    constraints=sud.constraints()
    sud.AC3(constraints)
    sud.AC3_table()
    sud.print_table()
    print()

    print("AFTER BACKTRACKING: ")
    sud.backtracking()
    sud.print_table()

    print("Total Execution Time: %s seconds"%(time.time()-start))

if __name__ == "__main__":
    main()

```

## 7.2 utilities.py

```

"""
-----
utilities.py
Utilities file for sudoku.py data structures
-----

CP468
Assignment 2
Authors: Keven Iskander, Carla Castaneda, Nicole Laslavic, Alexander Francis
__updated__ = "2020-11-02"
Queue implementation from CP164
-----
"""

from copy import deepcopy

class Queue:
    def __init__(self):
        """
        -----
        Initializes an empty queue. Data is stored in a list.
        Use: q = Queue()
        """

```

```

-----
Postconditions:
    Initializes an empty queue.
-----
"""
self._values = []
return

def is_empty(self):
    """
    -----
    Determines if the queue is empty.
    Use: b = q.is_empty()
    -----
    Postconditions:
        Returns True if the queue is empty, False otherwise.
    -----
    """
    return len(self._values) == 0

def is_full(self):
    """
    -----
    Determines if the queue is full.
    Use: b = q.is_full()
    -----
    Postconditions:
        Returns True if the queue is full, False otherwise.
    -----
    """
    return False

def __len__(self):
    """
    -----
    Returns the length of the queue.
    Use: n = len(q)
    -----
    Postconditions:
        Returns the number of values in the queue.
    -----
    """
    return len(self._values)

def insert(self, value):
    """
    -----
    Inserts a copy of value into the queue.
    Use: q.insert(value)
    -----
    Preconditions:
        value - a data element (?)
    Postconditions:

```

```

        a copy of value is added to the rear of the queue.
        -----
        """
        self._values.append(deepcopy(value))
        return

def remove(self):
    """
    -----
    Removes and returns value from the queue.
    Use: v = q.remove()
    -----
    Postconditions:
        returns
        value - the value at the front of the queue - the value is
        removed from the queue (?)
    -----
    """
    assert len(self._values) > 0, "Cannot remove from an empty queue"
    value = self._values.pop(0)
    return value

def peek(self):
    """
    -----
    Peeks at the front of queue.
    Use: v = q.peek()
    -----
    Postconditions:
        returns
        value - a copy of the value at the front of the queue -
        the value is not removed from the queue (?)
    -----
    """
    assert len(self._values) > 0, "Cannot peek at an empty queue"
    value = deepcopy(self._values[0])
    return value

def __iter__(self):
    """
    FOR TESTING ONLY
    -----
    Generates a Python iterator. Iterates through the queue
    from front to rear.
    Use: for v in q:
    -----
    Postconditions:
        returns
        value - the next value in the queue (?)
    -----
    """
    for value in self._values:
        yield value

```