Ex No.4: *Implementation and Analysis of DFS and BFS for an application*

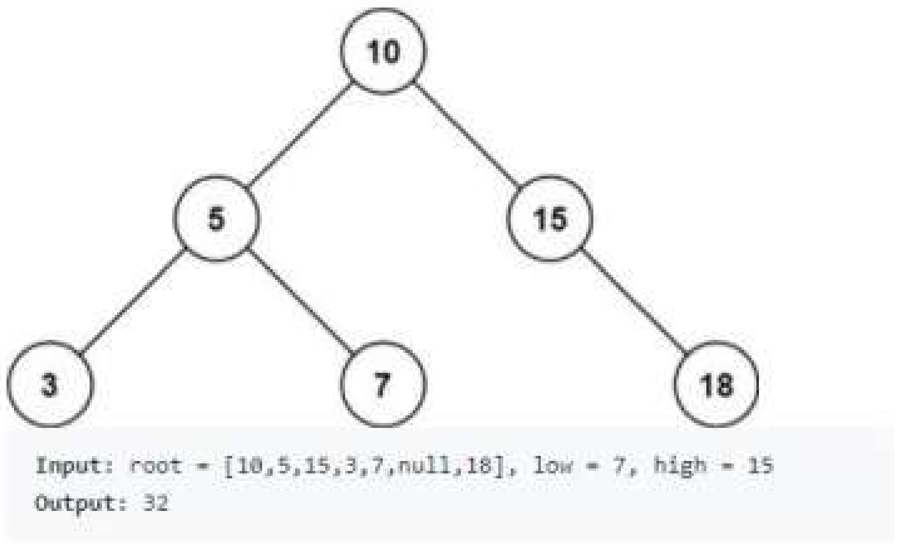
Aim :

1)Range Sum of Binary Tree

Given the root node of a binary search tree, return the sum of values of all nodes with a value in the range [low, high] using depth first and then breadth first search.

While :

- The number of nodes in the tree is in the range [1, 2 * 104].
- 1 <= Node.val <= 105
- 1 <= low <= high <= 105
- All Node.value are unique.



```
Input: root = [10,5,15,3,7,null,18], low = 7, high = 15
Output: 32
```

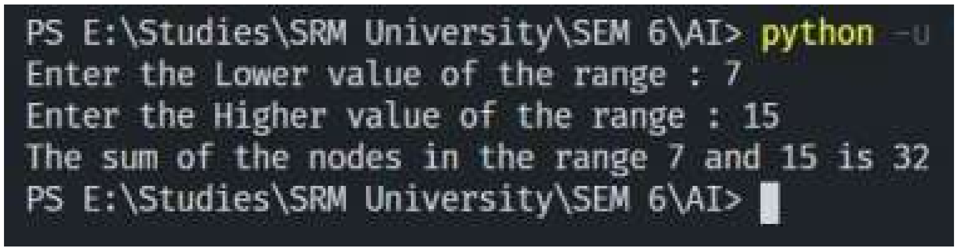
Algorithm :

1. We traverse the tree using a depth first search.
2. If node.value falls outside the range [L, R], (for example node.val < L), then we know that only the right branch could have nodes with value inside [L, R].
3. We showcase two implementations - one using a recursive algorithm, and one using an iterative one.
4. Time Complexity: O(N)O(N), where NN is the number of nodes in the tree. 5. Space Complexity: O(N)O(N)
5. For the recursive implementation, the recursion will consume additional space in the function call stack. In the worst case, the tree is of chain shape, and we will reach all the way down to the leaf node.
6. For the iterative implementation, essentially, we are doing a BFS (Breadth-First Search) traversal, where the stack will contain no more than two levels of the nodes. The maximal number of nodes in a binary tree is N/2.
7. Therefore, the maximal space needed for the stack would be O(N)O(N).

Depth First Search Code :

```
# ITERATIVE APPROACH
class Solution(object):
 def rangeSumBST(self, root, L, R):
 def dfs(node):
 if node:
 if L <= node.val <= R:
 self.ans += node.val
 if L < node.val:
26
 dfs(node.left)
 if node.val < R:
 dfs(node.right)
 self.ans = 0
 dfs(root)
 return self.ans
# RECURSIVE APPROACH
class Solution(object):
 def rangeSumBST(self, root, L, R):
 ans = 0
 stack = [root]
 while stack:
 node = stack.pop()
 if node:
 if L <= node.val <= R:
 ans += node.val
 if L < node.val:
 stack.append(node.left)
 if node.val < R:
 stack.append(node.right)
 return ans



bst = TreeNode(10)
bst.left = TreeNode(5)
bst.right = TreeNode(15)
bst.left.left = TreeNode(3)
bst.left.right = TreeNode(7)
bst.right.right = TreeNode(18)
min = int(input("Enter the Lower value of the range : "))
max = int(input("Enter the Higher value of the range : "))
sol = rangeSumBST(bst, min, max)
print(f"The sum of the nodes in the range {min} and {max} is {sol}")
```

Depth First Search Output :

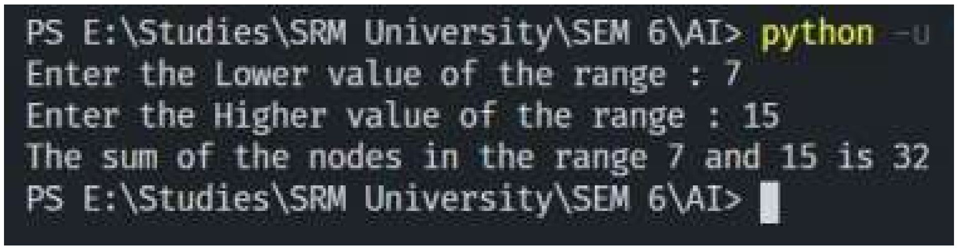


```
PS E:\Studies\SRM University\SEM 6\AI> python -u
Enter the Lower value of the range : 7
Enter the Higher value of the range : 15
The sum of the nodes in the range 7 and 15 is 32
PS E:\Studies\SRM University\SEM 6\AI>
```

Breadth First Code:

```
class Solution(object):
 def rangeSumBST(self, root, L, R):
 if root == None:
 return 0
 res = 0
 q = [root]
 while q:
 next = []
 for node in q:
 if L <= node.val <= R:
 res += node.val
 if node.left:
 next.append(node.left)
 if node.right:
 next.append(node.right)
 q = next
 return res
bst = TreeNode(10)
bst.left = TreeNode(5)
bst.right = TreeNode(15)
bst.left.left = TreeNode(3)
29
bst.left.right = TreeNode(7)
bst.right.right = TreeNode(18)
min = int(input("Enter the Lower value of the range : "))
max = int(input("Enter the Higher value of the range : "))
sol = rangeSumBST(bst, min, max)
print(f"The sum of the nodes in the range {min} and {max} is {sol}")
```

Breadth First Output:



```
PS E:\Studies\SRM University\SEM 6\AI> python -u
Enter the Lower value of the range : 7
Enter the Higher value of the range : 15
The sum of the nodes in the range 7 and 15 is 32
PS E:\Studies\SRM University\SEM 6\AI>
```

2)Implementation of Search in N-Queen Problem

The N–Queens problem is a classic problem that is often used in discussions of various search strategies. The problem is often defined in terms of a standard 8–by–8 chess board, although it can be defined for any N–by–N board and is solvable for N ³ 4.

The aim is to try solving N-Queen problem by Depth First Search (DFS) and Breadth First Search (BFS) algorithm.

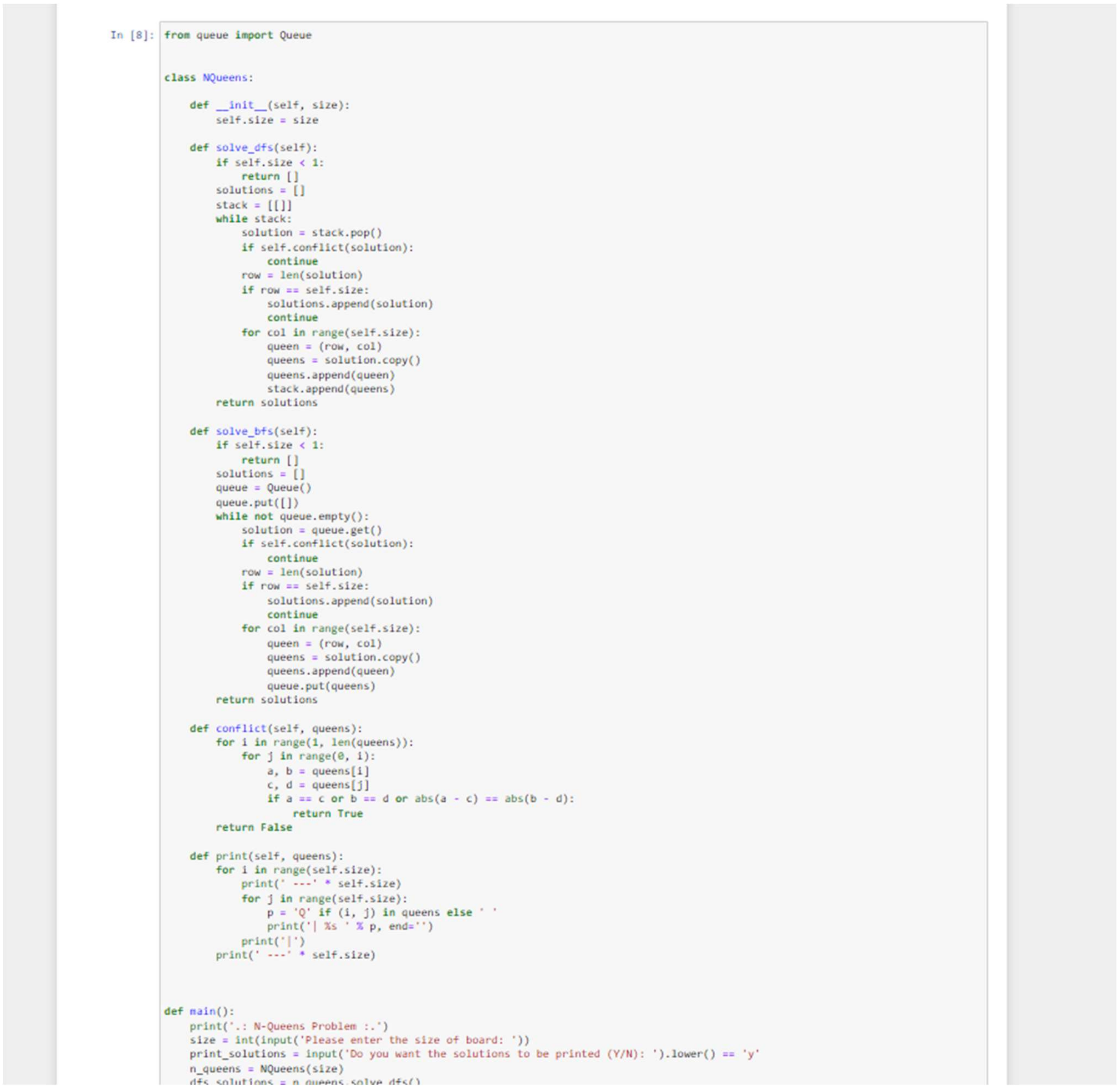
Code :

```
  from  queue import Queue


class NQueens:


                    def __init__(self, size):
                        self.size = size

                    def solve_dfs(self):
                        if self.size < 1:
                            return []
                        solutions = []
                        stack = [[]]
                        while stack:
                            solution = stack.pop()
                            if
                  self.conflict(solution):
                                continue
                            row = len(solution)
                            if row == self.size:

                  solutions.append(solution)
                                continue
                            for col in
                  range(self.size):
                                queen = (row, col)
                                queens =
                  solution.copy()

                  queens.append(queen)

                  stack.append(queens)
                        return solutions

                    def solve_bfs(self):
                        if self.size < 1:
                            return []
```

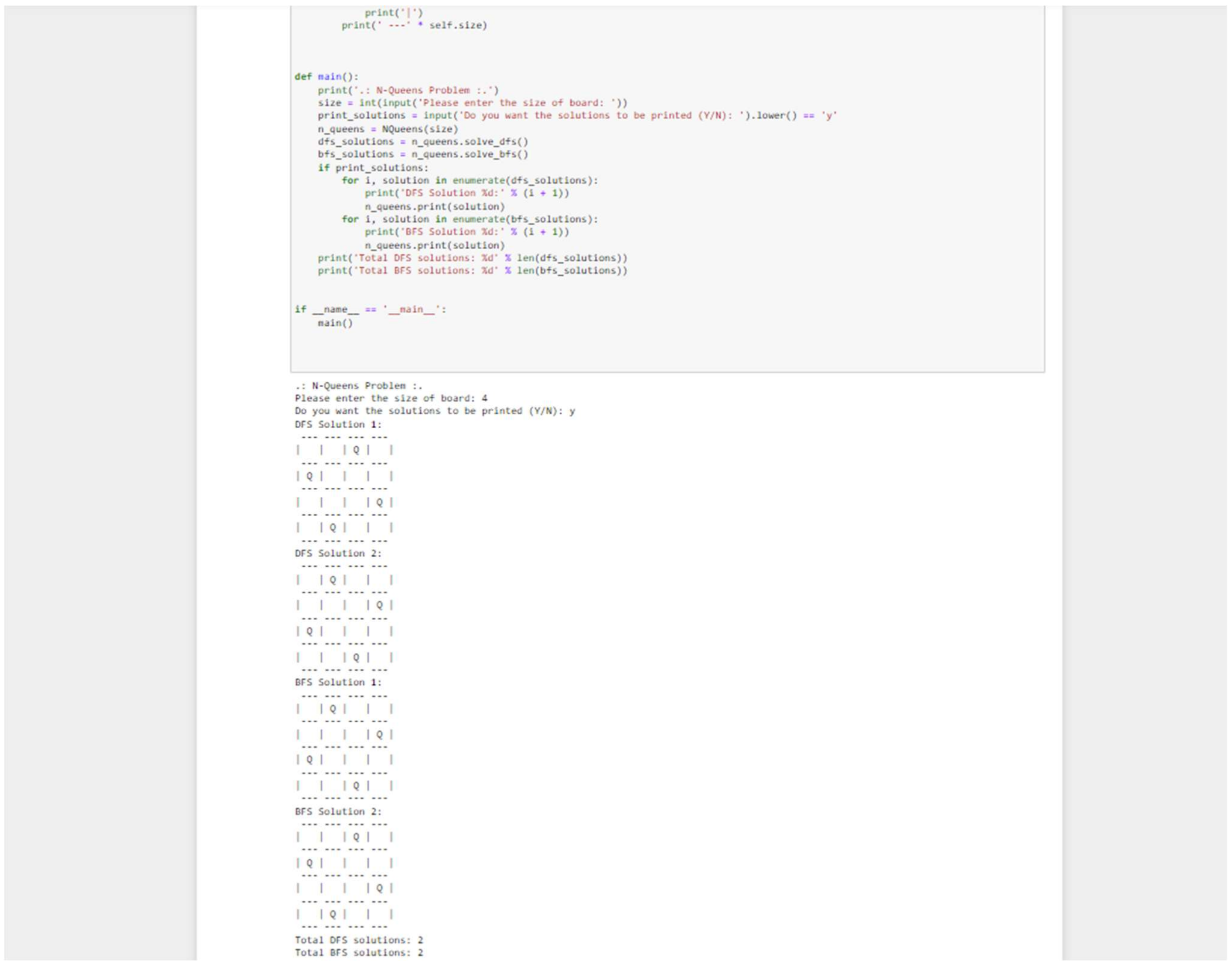
```
        solutions = []
        queue = Queue()
        queue.put([])
        while not queue.empty():
            solution = queue.get()
            if
self.conflict(solution):
                continue
            row = len(solution)
            if row == self.size:

solutions.append(solution)
                continue
            for col in
range(self.size):
                queen = (row, col)
                queens =
solution.copy()

queens.append(queen)
                queue.put(queens)
        return solutions

    def conflict(self, queens):
        for i in range(1,
len(queens)):
            for j in range(0, i):
                a, b = queens[i]
                c, d = queens[j]
                if a == c or b == d
or abs(a - c) == abs(b - d):
                    return True
        return False

    def print(self, queens):
        for i in range(self.size):
            print(' ---' * self.size)
            for j in
range(self.size):
                p = 'Q' if (i, j) in
queens else ' '
                print('| %s ' % p,
end='')
            print('|')
        print(' ---' * self.size)
```

```python
from n_queens import NQueens

def main():
    print('.: N-Queens Problem :.')
    size = int(input('Please enter the size of board: '))
    print_solutions = input('Do you want the solutions to be printed (Y/N): ').lower() == 'y'
    n_queens = NQueens(size)
    dfs_solutions = n_queens.solve_dfs()
    bfs_solutions = n_queens.solve_bfs()
    if print_solutions:
        for i, solution in enumerate(dfs_solutions):
            print('DFS Solution %d:' % (i + 1))
            n_queens.print(solution)
        for i, solution in enumerate(bfs_solutions):
            print('BFS Solution %d:' % (i + 1))
            n_queens.print(solution)
    print('Total DFS solutions: %d' % len(dfs_solutions))
    print('Total BFS solutions: %d' % len(bfs_solutions))


if __name__ == '__main__':
    main()
```

Output:



```
In [8]: from queue import Queue


        class NQueens:

            def __init__(self, size):
                self.size = size

            def solve_dfs(self):
                if self.size < 1:
                    return []
                solutions = []
                stack = [[]]
                while stack:
                    solution = stack.pop()
                    if self.conflict(solution):
                        continue
                    row = len(solution)
                    if row == self.size:
                        solutions.append(solution)
                        continue
                    for col in range(self.size):
                        queen = (row, col)
                        queens = solution.copy()
                        queens.append(queen)
                        stack.append(queens)
                return solutions

            def solve_bfs(self):
                if self.size < 1:
                    return []
                solutions = []
                queue = Queue()
                queue.put([])
                while not queue.empty():
                    solution = queue.get()
                    if self.conflict(solution):
                        continue
                    row = len(solution)
                    if row == self.size:
                        solutions.append(solution)
                        continue
                    for col in range(self.size):
                        queen = (row, col)
                        queens = solution.copy()
                        queens.append(queen)
                        queue.put(queens)
                return solutions

            def conflict(self, queens):
                for i in range(1, len(queens)):
                    for j in range(0, i):
                        a, b = queens[i]
                        c, d = queens[j]
                        if a == c or b == d or abs(a - c) == abs(b - d):
                            return True
                return False

            def print(self, queens):
                for i in range(self.size):
                    print(' ---' * self.size)
                    for j in range(self.size):
                        p = 'Q' if (i, j) in queens else ' '
                        print('| %s ' % p, end='')
                    print('|')
                print(' ---' * self.size)



        def main():
            print('.: N-Queens Problem :.')
            size = int(input('Please enter the size of board: '))
            print_solutions = input('Do you want the solutions to be printed (Y/N): ').lower() == 'y'
            n_queens = NQueens(size)
            dfs_solutions = n_queens.solve_dfs()
```



```python
            print('|')
        print(' ---' * self.size)



def main():
    print('.: N-Queens Problem :.')
    size = int(input('Please enter the size of board: '))
    print_solutions = input('Do you want the solutions to be printed (Y/N): ').lower() == 'y'
    n_queens = NQueens(size)
    dfs_solutions = n_queens.solve_dfs()
    bfs_solutions = n_queens.solve_bfs()
    if print_solutions:
        for i, solution in enumerate(dfs_solutions):
            print('DFS Solution %d:' % (i + 1))
            n_queens.print(solution)
        for i, solution in enumerate(bfs_solutions):
            print('BFS Solution %d:' % (i + 1))
            n_queens.print(solution)
    print('Total DFS solutions: %d' % len(dfs_solutions))
    print('Total BFS solutions: %d' % len(bfs_solutions))


if __name__ == '__main__':
    main()
```

```
.: N-Queens Problem :.
Please enter the size of board: 4
Do you want the solutions to be printed (Y/N): y
DFS Solution 1:
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---
DFS Solution 2:
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---
BFS Solution 1:
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---
BFS Solution 2:
 --- --- --- ---
|   |   | Q |   |
 --- --- --- ---
| Q |   |   |   |
 --- --- --- ---
|   |   |   | Q |
 --- --- --- ---
|   | Q |   |   |
 --- --- --- ---
Total DFS solutions: 2
Total BFS solutions: 2
```

Result :

Successfully found the sum of nodes in a binary search tree between any given range (min, max) and implemented the n-queen problem using both depth first search and breadth first search approach.