Searching and Sorting Algorithms

1. Linear Search

Aim:

To write a Python program to implement Linear Search and determine the time required to search for an element.

Algorithm:

Step 1: First, read the search element (Target element) in the array.

Step 2: In the second step compare the search element with the first element in the array.

Step 3: If both are matched, display "Target element is found" and terminate the Linear Search function.

Step 4: If both are not matched, compare the search element with the next element in the array.

Step 5: In this step, repeat steps 3 and 4 until the search (Target) element is compared with the last element of the array.

Step 6 - If the last element in the list does not match, the Linear Search Function will be terminated, and the message "Element is not found" will be displayed.

Program:

```
import time
import numpy as np
start = time.time()
def search(ls, x):
    for i in range(len(ls)):
        if ls[i] == x:
            return i
n = int(input("Enter number of elements : "))
a = list(map(int,input("\nEnter the elements : ").strip().split()))[:n]
key = int(input("Enter the element to search : "))
print("Your element is present at index ",search(a,key))
end = time.time()
ti = end-start
print("Time taken to search :",ti)
```

Output:

```
Enter number of elements : 5
Enter the elements : 3 4 5 6 1
Enter the element to search : 5
Your element is present at index  2
Time taken to search : 10.700574398040771
```

Result:

Thus the Python program to implement linear search is successfully executed and verified.

2. Binary Search using Recursive Function

Aim:

To write a Python program to implement recursive Binary Search and determine the time required to search an element.

Algorithm:

Step 1: Sort the array in ascending order.

Step 2: Set the low index to the first element of the array and the high index to the last element.

Step 3: Set the middle index to the average of the low and high indices.

Step 4: If the element at the middle index is the target element, return the middle index.

Step 5: If the target element is less than the element at the middle index, set the high index to the middle index – 1.

Step 6: If the target element is greater than the element at the middle index, set the low index to the middle index + 1.

Step 7: Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.

Program:

```
import time
start = time.time()
def BinarySearch(arr, k, low, high):
    if high >= low:
        mid = low + (high - low)//2
        if arr[mid] == k:
            return mid
        elif arr[mid] > k:
            return BinarySearch(arr, k, low, mid-1)
        else:
            return BinarySearch(arr, k, mid + 1, high)
    else:
        return -1
arr = [1, 3, 5, 7, 9]
print(arr)
k = int(input("enter the element at array:"))
result = BinarySearch(arr, k, 0, len(arr)-1)
if result != -1:
```

```
    print("Element is present at index " + str(result))
else:
    print("Not found")

BinarySearch(arr, k, 0, len(arr)-1)
end = time.time()
print("Time taken to search: ",start-end ,"seconds")
```

Output:

```
[1, 3, 5, 7, 9]
Enter an element from array: 5
Element is present at index 2
Time taken to search:  -4.114552736282349 seconds
```

Result:

Thus the Python program to implement Binary search is successfully executed and verified.

3. Pattern Searching

Aim:

To write a Python program to search given pattern from the given text and prints all occurrences.

Algorithm:

Step 1: KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as the size of the pattern) which is used to skip characters while matching.

Step 2: Name lps indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC”, and “ABC”.

Step 3: We search for lps in sub-patterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.

Step 4: For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

Step5: lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

Step 6: We calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index

Step 7: We initialize lps[0] and len as 0

Step 8:If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i].

Step 10:If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]

Step 11: See computeLPSArray () in the above code for details

Program:

```
def search(pat, txt):
        M = len(pat)
        N = len(txt)
        for i in range(N - M + 1):
                j = 0
                while(j < M):
```

```
                if (txt[i + j] != pat[j]):
                        break
                j += 1
        if (j == M):
                print("Pattern found at index ", i)
txt = input('Enter the text : ')
pat = input("Enter the pattern : ")
search(pat,txt)
```

Output:

```
Enter the text : banana
Enter the pattern : na
Pattern found at index  2
Pattern found at index  4
```

Result:

Thus the Python program to implement Pattern Search is successfully executed and verified.

4(a). Insertion Sort

Aim:

To write a Python program to sort a given set of elements using the Insertion sort method and determine the time required to sort the elements.

Algorithm:

Step 1 - If the element is the first element, assume that it is already sorted. Return

Step2 - Pick the next element, and store it separately in a key**.**

Step3 - Now, compare the key with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

Program:

```
import time
start =time.time()
def InsertionSort(a):
  for i in range(1, len(a)):
        temp = a[i]
        j = i-1
        while j >=0 and temp < a[j] :
              a[j+1] = a[j]
              j -= 1
        a[j+1] = temp
a = [10, 5, 13, 8, 2]
print("Before after sorting:")
print(a)
InsertionSort(a)
print("Array after sorting:")
print(a)
end =time.time()
print("Time taken for searching",end-start,"seconds")
```

Output:

Before after sorting:
[10, 5, 13, 8, 2]
Array after sorting:
[2, 5, 8, 10, 13]
Time taken for searching 0.1542215347290039 seconds

Result:

Thus the Python program to implement Insertion sort is successfully executed and verified.

4(b). Heap Sort

Aim:

To write a Python program to sort a given set of elements using the Heap sort method and determine the time required to sort the elements.

Algorithm:

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Min Heap.**

Step 3 - Delete the root element from Min Heap using **Heapify** method.

Step 4 - Put the deleted element into the Sorted list.

Step 5 - Repeat the same until Min Heap becomes empty.

Step 6 - Display the sorted list.

Program:

```
import time
start = time.time()
def heapify(array, a, b):
        largest = b
        l = 2 * b + 1
        root = 2 * b + 2
        if l < a and array[b] < array[l]:
                largest = l
        if root < a and array[largest] < array[root]:
                largest = root
        # Change root
        if largest != b:
           array[b], array[largest] = array[largest], array[b]
           heapify(array, a, largest)
# sort an array of given size
def Heap_Sort(array):
        a = len(array)
        # maxheap..
        for b in range(a // 2 - 1, -1, -1):
                heapify(array, a, b)
        # extract elements
```

```
        for b in range(a-1, 0, -1):
                array[b], array[0] = array[0], array[b] # swap
                heapify(array, b, 0)
# Driver code
array = [ 7, 2, 5, 6, 3, 1, 8, 4]
print("The original array is:\n ", array)

print(start)
Heap_Sort(array)
#end= time.time()
a = len(array)
print ("Array after sorting is: \n", array)
end= time.time()
def measure_heap_sort_time(array):
    start_time = time.time()
    Heap_sort(array)
    end_time = time.time()
    return end_time - start_time

print("The required time is:",float(end-start))
```

Output:

```
The original array is:
  [7, 2, 5, 6, 3, 1, 8, 4]
Array after sorting is:
 [1, 2, 3, 4, 5, 6, 7, 8]
The required time is: 0.14311861991882324
```

Result:

Thus the Python program to implement Heap sort is successfully executed and verified.

Graph Algorithms

5. Breadth First Search

Aim:

To write a Python program to develop a program to implement graph traversal using Breadth First Search.

Algorithm:

Bfs(self,start)

//graph traversal using Breadth First Search

//inputs: graph with n edges

//outputs: BFS of the graph

Step 1: Start by putting any one of the graph's vertices at the back of a queue.

Step 2: Take the front item of the queue and add it to the visited list.

Step 3: Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.

Step 4: Keep repeating steps 2 and 3 until the queue is empty.

Program:

```
from collections import deque

class Graph:

    def _init_(self):

        self.adjacency_list = {}

    def add_edge(self, u, v):

        if u not in self.adjacency_list:

            self.adjacency_list[u] = []

        if v not in self.adjacency_list:

            self.adjacency_list[v] = []

        self.adjacency_list[u].append(v)
```

```
        self.adjacency_list[v].append(u)
    def bfs(self, start):
        visited = {start}
        queue = deque([start])
        while queue:
            vertex = queue.popleft()
            print(vertex, end=" ")
            for neighbour in self.adjacency_list[vertex]:
                if neighbour not in visited:
                    visited.add(neighbour)
                    queue.append(neighbour)
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)
g.add_edge(4, 5)print("The BFS of the graph is:")
g.bfs(2)
```

Output:

```
The BFS of the graph is:
0 1 2 3 4 5
```

Result:

Thus the Python program to implement graph traversal using Breadth First Search is successfully executed and verified.

6. Depth First Search

Aim:

To write a Python program to develop a program to implement graph traversal using Depth First Search.

Algorithm:

Dfs(self,start)

//graph traversal using depth First Search

//inputs: graph with n edges

//outputs: DFS of the graph

Step 1: Start by putting any one of the graph's vertices on top of a stack.

Step 2: Take the top item of the stack and add it to the visited list.

Step 3: Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.

Step 4: Keep repeating steps 2 and 3 until the stack is empty.

Program:

```
class Graph:
    def _init_(self):
        self.adjacency_list = {}
    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        if v not in self.adjacency_list:
            self.adjacency_list[v] = []
        self.adjacency_list[u].append(v)
        self.adjacency_list[v].append(u)
    def dfs_util(self, vertex, visited):
```

```
        visited.add(vertex)
        print(vertex, end=" ")
        for neighbour in self.adjacency_list[vertex]:
            if neighbour not in visited:
                self.dfs_util(neighbour, visited)
    def dfs(self, start):
        visited = set()
        self.dfs_util(start, visited)
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 3)
g.add_edge(1, 4)
g.add_edge(2, 5)
g.add_edge(4, 5)
print("The DFS of the graph is:")
g.dfs(2)
```

Output:

```
The DFS of the graph is:
0 1 3 4 5 2
```

Result:

Thus the Python program to implement graph traversal using Depth First Search is successfully executed and verified.

7. Dijkstra's algorithm

Aim:

To write a Python program to find the shortest paths to other vertices using Dijkstra's algorithm.

Algorithm:

dijkstra(graph, start):

//shortest paths to using Dijkstra's algorithm

//inputs: graph with n edges

//outputs: shortest path to other vertices from vertex

Step 1: Initialize the distance to all vertices as infinity, except the starting vertex.

Step 2: Create a priority queue and add the starting vertex with distance 0.

Step 3: Process vertices in the queue until all vertices have been visited.

Step 4: Remove the vertex with the shortest distance from the queue.

Step 5: If the distance to the current vertex is greater than the known distance, skip it.

Step 6: For each neighbor of the current vertex, calculate the distance to it through the current vertex.

Step 7: If the calculated distance is shorter than the known distance, update the distance.

Step 8: Add the neighbor to the queue with the updated distance.

Step 9: Return the distances to all vertices.

Program:

```
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    heap = [(0, start)]
```

```
    while heap:
        (current_distance, current_vertex) = heapq.heappop(heap)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))
    return distances
graph = {'A': {'B': 2, 'C': 4},
         'B': {'D': 3},
         'C': {'D': 1, 'E': 5},
         'D': {'E': 1},
         'E': {}
        }
start_vertex = 'A'
shortest_distances = dijkstra(graph, start_vertex)
print("The shortest distance using Dijkstra algorithm:\n",shortest_distances)
```

Output:

```
The shortest distance using Dijkstra algorithm:
{'A': 0, 'B': 2, 'C': 4, 'D': 5, 'E': 6}
```

Result:

Thus the Python program to find the shortest paths to other vertices using Dijkstra's algorithm is successfully executed and verified.

8. Prim's Algorithm

Aim:

To write a Python program to find the minimum cost spanning tree of a given undirected graph using Prim's algorithm.

Algorithm:

Prim(graph):

//To find minimum spanning tree using prim's algorithm

//inputs: graph

//output: minimum spanning tree

Step 1: Initialize the set of visited vertices and the minimum spanning tree.

Step 2: Choose a starting vertex arbitrarily and mark it as visited.

Step 3: Add the edges of the starting vertex to the priority queue.

Step 4: Process edges in the queue until all vertices have been visited.

Step 5: Remove the edge with the smallest weight from the queue.

Step 6: If both vertices have already been visited, skip the edge.

Step 7: Add the edge to the minimum spanning tree and mark the new vertex as visited.

Step 8: Add the edges of the new vertex to the priority queue.

Step 9: Return the minimum spanning tree.

Program:

```
import heapq

def prim(graph):
    visited = set()
    minimum_spanning_tree = []
    start_vertex = list(graph.keys())[0]
    visited.add(start_vertex)
```

```
    edges = [(weight, start_vertex, neighbor) for neighbor, weight in
graph[start_vertex].items()]
    heapq.heapify(edges)
    while edges:
        (weight, vertex1, vertex2) = heapq.heappop(edges)
        if vertex1 in visited and vertex2 in visited:
            continue
        minimum_spanning_tree.append((vertex1, vertex2, weight))
        visited.add(vertex1 if vertex1 not in visited else vertex2)
        for neighbor, weight in graph[vertex1 if vertex1 not in visited else
vertex2].items():
            if neighbor not in visited:
                heapq.heappush(edges, (weight, vertex1 if vertex1 not in visited else
neighbor, vertex2 if vertex2    not in visited else neighbor))
    return minimum_spanning_tree
graph={'A': {'B': 2, 'C': 4},
            'B': {'A': 2, 'D': 3},
            'C': {'A': 4, 'D': 1, 'E': 5},
            'D': {'B': 3, 'C': 1, 'E': 1},
            'E': {'C': 5, 'D': 1}
           }
min_spanning_tree=prim(graph)
print("The minimum spanning tree using prim's algorithm is:\n",min_spanning_tree)
```

Output:

The minimum spanning tree using prim's algorithm is:

[('A', 'B', 2), ('D', 'D', 3), ('C', 'C', 1), ('E', 'E', 1)]

Result:

Thus the Python program to find the minimum cost spanning tree of a given undirected graph using Prim’s algorithm is successfully executed and verified.

9. Floyd's algorithm

Aim:

To write a Python program to find the All Pair Shortest Path using Floyd's algorithm.

Algorithm:

floyd(graph):

//To find all-pairs-shortest-paths using Floyd algorithm

//inputs: graph in 2D matrix

//output: all-pair-shortest-path

Step 1: Initialize the distance matrix

Step 2: Fill in the distance matrix with the direct distances from the graph.

Step 3: Compute the shortest path for each pair of vertices.

Step 4: Return the distance matrix.

Program:

```
def floyd(graph):
    distance_matrix = [[float('inf') if i != j else 0 for j in range(len(graph))] for i in range(len(graph))]
    for i in range(len(graph)):
        for j in range(len(graph)):
            if graph[i][j] != 0:
                distance_matrix[i][j] = graph[i][j]
    for k in range(len(graph)):
        for i in range(len(graph)):
            for j in range(len(graph)):
                if distance_matrix[i][k] + distance_matrix[k][j] < distance_matrix[i][j]:
                    distance_matrix[i][j] = distance_matrix[i][k] + distance_matrix[k][j]
```

```
    print("The shortest distance matrix is:\n",distance_matrix)
I = float('inf')
adjMatrix = [[0, I, -2, I],
             [4, 0, 3, I],
             [I, I, 0, 2],
             [I, -1, I, 0]
            ]
floyd(adjMatrix)
```

Output:

The shortest distance matrix is:

[[0, -1, -2, 0], [4, 0, 2, 4], [5, 1, 0, 2], [3, -1, 1, 0]]

Result:

Thus the Python program to find the All Pair Shortest Path using floyd's algorithm is successfully executed and verified.

10. Transitive closure using Warshall's algorithm

Aim:

To write a Python program to Compute the transitive closure of a given directed graph using Warshall's algorithm.

Algorithm:

Warshall(A[1...n, 1...n]) // A is the adjacency matrix
$R^{(0)} \leftarrow A$
for k ← 1 to n do
for i ← 1 to n do
for j ← to n do
$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or ($R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j]$)
return $R^{(n)}$

Program:

```
from collections import defaultdict

class Graph:

        def __init__(self, vertices):

                self.V = vertices

        def printSolution(self, reach):

                print ("Following matrix transitive closure of the given graph ")

                for i in range(self.V):

                        for j in range(self.V):

                                if (i == j):

                                    print ("%7d\t" % (1),end=" ")

                                else:

                                    print ("%7d\t" %(reach[i][j]),end=" ")

                        print()

        def transitiveClosure(self,graph):
```

```
        reach =[i[:] for i in graph]
        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):
                    reach[i][j] = reach[i][j] or (reach[i][k] and
reach[k][j])
        self.printSolution(reach)
g= Graph(4)
graph = [[1, 1, 0, 1],
        [0, 1, 1, 0],
        [0, 0, 1, 1],
        [0, 0, 0, 1]]
g.transitiveClosure(graph)
```

Output:

Following matrix transitive closure of the given graph

1 1 1 1

0 1 1 1

0 0 1 1

0 0 0 1

Result:

Thus the Python program to Compute the transitive closure of a given directed graph using Warshall's algorithm is successfully executed and verified.

Algorithm Design Techniques

11. Maximum and Minimum numbers in a given list

Aim:

To write a Python program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique.

Algorithm:

FindMinMax(low,high,arr)

//Finding maximum and minimum using Divide-and-Conquer

//inputs: array of size n

//outputs: max and min elements of array

Step 1:Divide the array into two halves.

Step 2:Recursively find the minimum and maximum values in each half.

Step 3:Compare the minimum and maximum values found in each half to find the overall minimum and maximum values.

Step 4:Return the minimum and maximum values found in step 3.

Program:

```
def getMinMax(low, high, arr):
        arr_max = arr[low]
        arr_min = arr[low]
        if low == high:
                arr_max = arr[low]
                arr_min = arr[low]
                return (arr_max, arr_min)
        elif high == low + 1:
                if arr[low] > arr[high]:
                        arr_max = arr[low]
```

```
                arr_min = arr[high]
        else:
                arr_max = arr[high]
                arr_min = arr[low]
        return (arr_max, arr_min)
    else:
        mid = int((low + high) / 2)
        arr_max1, arr_min1 = getMinMax(low, mid, arr)
        arr_max2, arr_min2 = getMinMax(mid + 1, high, arr)
    return (max(arr_max1, arr_max2), min(arr_min1, arr_min2))
# Driver code
arr = [1000, 11, 445, 1, 330, 3000]
high = len(arr) - 1
low = 0
arr_max, arr_min = getMinMax(low, high, arr)
print('Minimum element is ', arr_min)
print('Maximum element is ', arr_max)
```

Output:

Minimum element is 1

Maximum element is 3000

Result:

Thus the python program to find out the maximum and minimum numbers in a given list of n numbers using the divide and conquer technique is successfully executed and verified.

12(a) Merge Sort

Aim:

To write a Python program to implement Merge sort to sort an array of elements and to determine the time required to sort.

Algorithm:

Merge_sort(A[0..n − 1])

//Sorts array A[0..n − 1] by recursive merge sort

//Input: An array A[0..n − 1] of orderable elements

//Output: Array A[0..n − 1] sorted in non-decreasing order

if n>1

Copy A[0...(n/2)] to B[0...(n/2)]

Copy A[(n/2+1) ...n-1] to C[0...(n-1)/2]

Merge sort (B[0..(n/2)]

Merge sort (C[0..(n-1)/2]

Merge (B,C,A)

Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])

//Merges two sorted arrays into one sorted array

//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted

//Output: Sorted array A[0..p + q − 1] of the elements of B and C

i ←0; j ←0; k←0

while i <p and j <q do

if B[i]≤ C[j]

A[k]←B[i]; i ←i + 1

else

A[k]←C[j]; j ←j + 1

k←k + 1

if i = p

copy C[j..q − 1] to A[k..p + q − 1]

else

copy B[i..p − 1] to A[k..p + q − 1]

Program:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    # Split the array in half
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    # Recursively sort the left and right halves
    left = merge_sort(left)
    right = merge_sort(right)
    # Merge the sorted left and right halves
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
```

```
    result.extend(left[i:])
    result.extend(right[j:])
    return result
a=[20,12,34,10,52,19]
print("The sorted array using merge sort: ",merge_sort(a));
```

Output:

The sorted array using merge sort: 10,12,19,20,34,52

Result:

Thus the Python program to Implement Merge sort to sort an array of elements is successfully executed and verified.

12(b) Quick sort

Aim:

To write a Python program to implement Quick sort to sort an array of elements and to determine the time required to sort.

Algorithm:

Quick_sort(A[l…r])

```
//sorts a sub array by quick sort
//inputs: subarray of array[0….n-1] defined by its left and right indices l and r
//output: subarray A[l…r] sorted in non-descending order
If p<r
        q=Partition(A,l,r)
        Quick_sort(A,l,q-1)
        Quick_sort(A,q+1,r)
```

Partition(A[l..r])

```
//partitions a subarray using hoare's algorithm using first element as pivot element
//inputs: subarray of array[0….n-1] defined by its left and right indices l and r
//output: Partition of A[l..r] with the split position turned as this function's value
P←A[l]
i←l , j←r+1
repeat
        repeat i = i+1 until A[i]>=p
        repeat j= j-1 until A[j]<=p
        swap(A[i],A[j})
until i>=j
swap(A[i],A[j])
swap(A[l],A[j])
return j
```

Program:

```
def quick_sort(arr):

    if len(arr) <= 1:

        return arr

    pivot = arr[-1]

    less = []
```

```
    greater = []
    equal = []
    for element in arr:
        if element < pivot:
            less.append(element)
        elif element > pivot:
            greater.append(element)
        else:
            equal.append(element)
        sorted_less = quick_sort(less)
    sorted_greater = quick_sort(greater)
    return sorted_less + equal + sorted_greater
a=[20,12,34,10,52,19]
print("The sorted array using Quick sort: ",quick_sort(a));
```

Output:

The sorted array using Quick sort: 10,12,19,20,34,52

Result:

Thus the Python program to Implement Quick sort to sort an array of elements is successfully executed and verified.

State Space Search Algorithms

13. Implement N Queens problem using Backtracking

Aim:

To write a Python program to implement N Queens problem using Backtracking.

Algorithm:

Step 1: Initialize an empty chessboard of size NxN.

Step 2: Start with the leftmost column and place a queen in the first row of that column.

Step 3: Move to the next column and place a queen in the first row of that column.

Step 4: Repeat step 3 until either all N queens have been placed or it is impossible to place a queen in the current column without violating the rules of the problem.

Step 4: If all N queens have been placed, print the solution.

Step 5: If it is not possible to place a queen in the current column without violating the rules of the problem, backtrack to the previous column.

Step 6: Remove the queen from the previous column and move it down one row.

Step 7: Repeat steps 4-7 until all possible configurations have been tried.

Program:

```
global N

N = 4

def printSolution(board):
        for i in range(N):
                for j in range(N):
                        print(board[i][j], end = " ")
                print()

def isSafe(board, row, col):
        for i in range(col):
                if board[row][i] == 1:
```

```
                return False
        for i, j in zip(range(row, -1, -1),range(col, -1, -1)):
                if board[i][j] == 1:
                        return False
        for i, j in zip(range(row, N, 1),range(col, -1, -1)):
                if board[i][j] == 1:
                        return False
        return True
def solveNQUtil(board, col):
        if col >= N:
                return True
        for i in range(N):
                if isSafe(board, i, col):
                        board[i][col] = 1
                        if solveNQUtil(board, col + 1) == True:
                                return True
                        board[i][col] = 0
        return False
def solveNQ():
        board = [ [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0] ]
        if solveNQUtil(board, 0) == False:
                print ("Solution does not exist")
                return False
```

```
        printSolution(board)
        return True
# Driver Code
solveNQ()
```

Output:

```
0   0   1   0
1   0   0   0
0   0   0   1
0   1   0   0
```

Result:

Thus the Python program to Implement N Queens problem using Backtracking is successfully executed and verified.

14. Finding the k^{th} smallest number using randomized algorithm

Aim:

To write a Python program to Implement randomized algorithms for finding the k^{th} smallest number.

Algorithm:

Step 1: check if k>0&& k<=r-l+1:

- declare pos as randomPartition(arr,l,r).
- check if pos-1==k-1 than return arr[pos].
- check if pos-1>k-1 than recursively call kthsmallest(arr,l,pos-1,k).
- return recursively call kthsmallest(arr,pos+1,r,k-pos+l-1).

Step 2: return INT_MAX.

Program:

```
import random

def kthSmallest(arr, l, r, k):
    if (k > 0 and k <= r - l + 1):
        pos = randomPartition(arr, l, r)
        if (pos - l == k - 1):
            return arr[pos]
        if (pos - l > k - 1):
            return kthSmallest(arr, l, pos - 1, k)
        return kthSmallest(arr, pos + 1, r,k - pos + l - 1)
    return 999999999999

def swap(arr, a, b):
    temp = arr[a]
```

```
        arr[a] = arr[b]
        arr[b] = temp
def partition(arr, l, r):
        x = arr[r]
        i = l
        for j in range(l, r):
                if (arr[j] <= x):
                        swap(arr, i, j)
                        i += 1
        swap(arr, i, r)
        return i
def randomPartition(arr, l, r):
        n = r - l + 1
        pivot = int(random.random() * n)
        swap(arr, l + pivot, r)
        return partition(arr, l, r)
# Driver Code
if __name__ == '__main__':
        arr = [12, 3, 5, 7, 4, 19, 26]
        n = len(arr)
        k = 3
        print("K'th smallest element is", kthSmallest(arr, 0, n - 1, k));
```

Output:

K'th smallest element is 5

Result:

Thus the Python program to Implement randomized algorithms for finding the k^{th} smallest number is successfully executed and verified.