Practical 1

Aim: Analyse Selection Sort algorithm on different input of size. Use random function to generate input sequence.

Theory:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Algorithm:

```
SELECTION SORT (ARR, N)
Step 1: Repeat Steps 2 and 3 for K = 0 to N-1
Step 2: CALL SMALLEST (ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]
[END OF LOOP]
Step 4: EXIT

SMALLEST (ARR, K, N, POS)
Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K
Step 3: Repeat for J = K+1 to N-1
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
Step 4: RETURN POS
```

Complexity of Selection Sort Algorithm:

To sort an unsorted list with '**n**' number of elements we need to make **((n-1) +(n-2) +(n-3) +.....+1) = (n (n-1))/2** number of comparisons in the worst case. If the list already sorted, then it requires '**n**' number of comparisons.

Worst Case: $O(n^2)$

Best Case: $\Omega(n^2)$

Average Case: $\Theta(n^2)$

Worst Case(Space): O(1)

Program: Implementation of algorithm with a set of data in **Python.**

```
from time import time
from random import randrange
import matplotlib.pyplot as plt
from numpy import linspace

MAX, size = int(5e3), 1
elements, times = [], []

while(size < MAX):
    size *= 2
    arr = [0] * size
    for i in range(size):
        arr[i] = randrange(0, size)
    start = time()
    # Selection Sort
    for i in range(size - 1):
        mi = i
        for j in range(i+1, size):
            if(arr[j] < arr[mi]):
                mi = j
        arr[i], arr[mi] = arr[mi], arr[i]

    end = time()

    total_time  = (end - start) * 1000
    elements.append(size)
    times.append(total_time)
    print(f"Time taken to sort array of
{size:.0E} numbers is
    {total_time:.3f} Milliseconds")
```

For Plotting Graph:

```
x = linspace(0, max(elements), 100)
y = (x/100) ** 2
plt.xlabel('Number Count(N)')
plt.ylabel('Time(in Milliseconds)')
plt.plot(x, y, label ='N^2', color =
      'orange')
plt.plot(elements, times, label
    ='Selection Sort', color = 'blue')
plt.grid()
plt.legend()
plt.show()
```

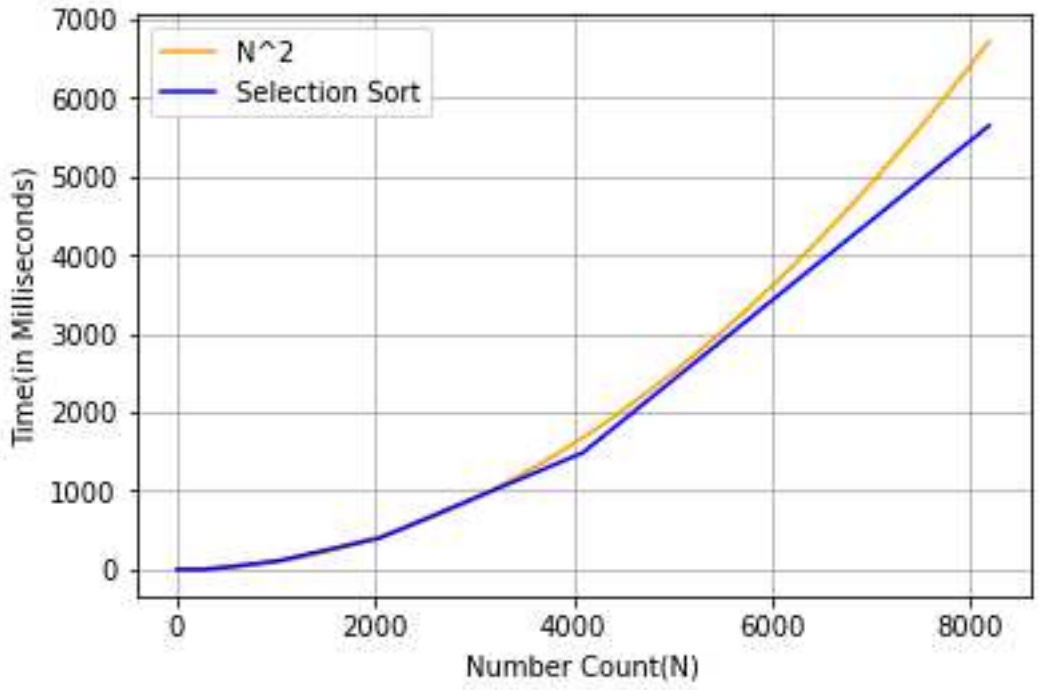
Analysis:

Program to be executed for various sizes of input.

Sr. No.	Size of input(n)	Time Taken (Milliseconds)
1	2000	406.236
2	2125	484.409
3	2250	515.629
4	2375	562.488
5	2500	646.050
6	2625	718.732
7	2750	758.654
8	2875	864.136
9	3000	894.991
10	3125	1086.043
11	3250	1031.282
12	3375	1140.610
13	3500	1244.066
14	3625	1428.292
15	3750	1463.008
16	3875	1499.617
17	4000	1617.408
18	4125	1703.092
19	4250	1829.287
20	4375	1906.251
21	4500	2031.253
22	4625	2140.624
23	4750	2234.373
24	4875	2484.395
25	5000	2620.834

Conclusion:

Selection Sort's worst case time Complexity can be visualized using the following graph.



Maximum Time Taken by Selection sort is bounded by N^2.

Practical 2

Aim: Analyse Merge Sort Sorting algorithm on different input of size. Use random function to generate input sequence

Theory:

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Using the Divide and Conquer technique, we divide a problem into sub problems. When the solution to each sub problem is ready, we 'combine' the results from the sub problems to solve the main problem.

Suppose we had to sort an array A. A sub problem would be to sort a sub-section of this array starting at

index `p` and ending at index `r`, denoted as `A[p..r]`.

Divide
If q is the half-way point between p and r, then we can split the subarray `A[p..r]` into two arrays `A[p..q]` and `A[q+1, r]`.

Conquer
In the conquer step, we try to sort both the subarrays `A[p..q]` and `A[q+1, r]`. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.
Combine
When the conquer step reaches the base step and we get two sorted subarrays `A[p..q]` and `A[q+1, r]` for array `A[p..r]`, we combine the results by creating a sorted array `A[p..r]` from two sorted subarrays `A[p..q]` and `A[q+1, r]`.

Algorithm(Merge Sort):

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide
     the array into two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in
       step 2 and 3:
             Call merge(arr+l, arr+r)
```

Algorithm(Merge):

```
merge(A, B) is
    inputs A, B : list
    returns list
    C := new empty list
    while A is not empty and B is not empty do
        if head(A) ≤ head(B) then
            append head(A) to C
            drop the head of A
        else
            append head(B) to C
            drop the head of B
    while A is not empty do
        append head(A) to C
        drop the head of A
    while B is not empty do
        append head(B) to C
        drop the head of B
    return C
```

Complexity of Selection Sort Algorithm:

Merge sort always divides the array in two halves and take linear time to merge two halves.

It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.

Worst Case: O(Nlog(N))

Best Case: Ω(Nlog(N))

Average Case: Θ(Nlog(N))

Worst Case(Space): O(N)

Program: Implementation of algorithm with a set of data in **Python.**

```
from time import time
from random import randrange
import matplotlib.pyplot as plt
from numpy import linspace
from math import log2

MAX, size = int(1e5), 1
elements, times = [], []


def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r- m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0 , n1):
        L[i] = arr[l + i]
    for j in range(0 , n2):
        R[j] = arr[m + 1 + j]
    i = 0
    j = 0
    k = l
    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
```

```
def mergeSort(arr,l,r):
    if l < r:
        m = (l+(r-1))//2
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)


while(size < MAX):
    size *= 2
    arr = [0] * size
    for i in range(size):
        arr[i] = randrange(0, size)
    start = time()
    # Merge Sort
    mergeSort(arr, 0, len(arr) - 1)

    end = time()
    total_time  = (end - start) * 1000
    elements.append(size)
    times.append(total_time)
    print(f"Time taken to sort array of
    {size:.0E} numbers is {total_time:.3f}
      Milliseconds")
```

For Plotting Graph:

```
x = linspace(0, max(elements), 100)
y = (x/1000) ** 2
plt.plot(x, y, label ='N^2', color = 'Red')
x = linspace(0, max(elements), 100)
y = [0] * 100
for i in range(1, len(x)):
    y[i] = (x[i]/500 * (log2(x[i]/500)))
plt.plot(x, y, label ='Nlog(N)', color = 'orange')
plt.xlabel('Number Count')
plt.ylabel('Time(in Milliseconds)')
plt.plot(elements, times, label ='Mege Sort', color =
'blue')
plt.grid()
plt.legend()
plt.show()
```

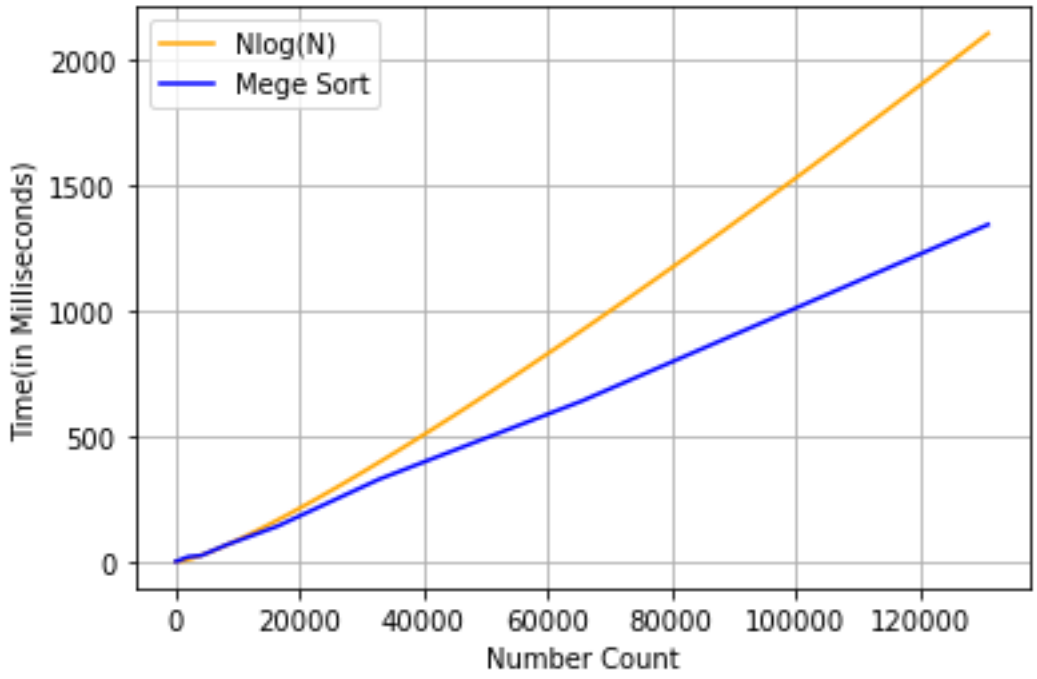
Analysis:

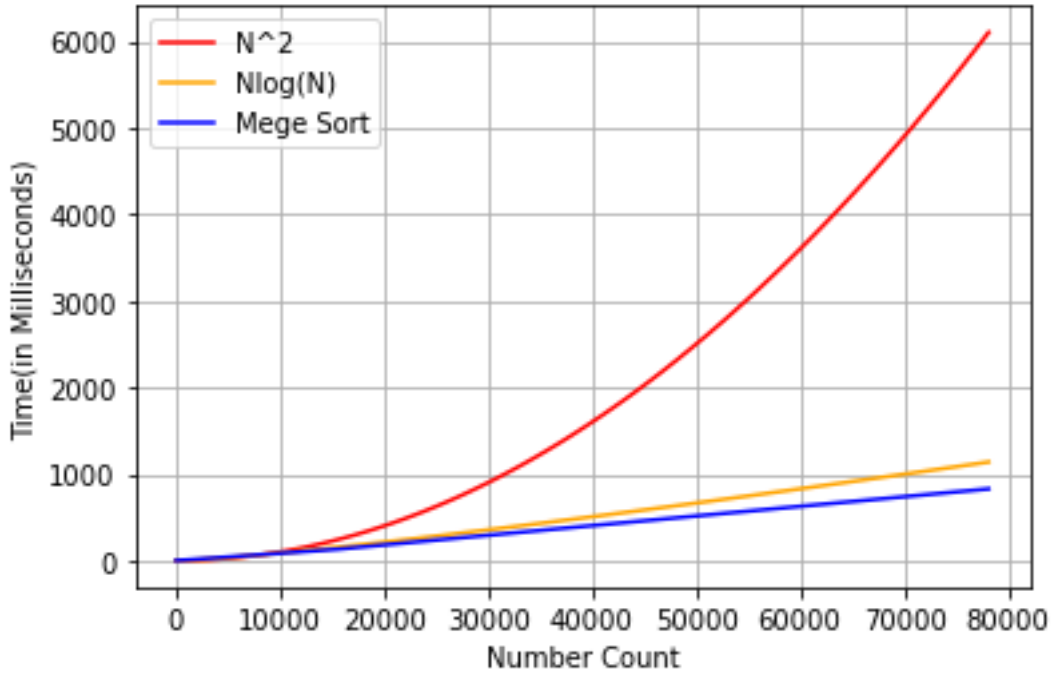
Program to be executed for various sizes of input.

Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	10000	78.134
2	10125	109.607
3	10250	78.112
4	10375	78.125
5	10500	93.748
6	10625	78.123
7	10750	93.752
8	10875	109.370
9	11000	93.779
10	11125	108.944
11	11250	109.349
12	11375	93.742
13	11500	93.750
14	11625	109.380
15	11750	109.361
16	11875	101.978
17	12000	109.397
18	12125	109.384
19	12250	93.661
20	12375	124.990
21	12500	109.392
22	12625	133.039
23	12750	109.113
24	12875	115.504
25	13000	108.32

Conclusion:

Merge Sort worst case time Complexity can be visualized by the following graphs.





Maximum Time Taken by Merge Sort is bounded by Nlog(N).

Practical 3

Aim: Analyse Linear Search and Binary Search Algorithm on different input sizes.

Theory:

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

Linear search is mostly used to search an unordered list of elements (array in which and initialized as, data elements are not sorted). For example, if an array A[10] is declared

```
int A[10] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
```

Val = 7 then Pos = 3

Binary search is a searching algorithm that works efficiently with a sorted list.

The mechanism of binary search can be better understood by an analogy of a telephone directory. When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory. Again, we open some page in the middle and the whole process is repeated until we finally find the right name. The same mechanism is applied in the binary search.

Binary Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Algorithm(Linear Search):

```
Linear Search ( Array arr, Value
to_search)

Step 1: Set found = False
Step 2: Repeat step 3 while arr has
elements
Step 3: if A[i] = to_search then set
found = True and go to step 6
Step 4: EXIT
```

Algorithm (Binary Search):

```
Step 1: [INITIALIZE] SET BEG =
lower_bound
END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while
BEG <=END
Step 3: SET MID = (BEG + END)/2
Step 4: IF A[MID] = VAL
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
Step 5: IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE
ARRAY"
[END OF IF]
Step 6: EXIT
```

Complexity of Linear Search Algorithm:

Linear Search takes linear time to search the List.

It requires O(1) additional space.

Worst Case: O(N)

Best Case: Ω(1)

Average Case: θ(N/2)

Worst Case(Space): O(1)

Complexity of Binary Search Algorithm:

Binary Search cuts the search space in half in each step.

It requires O(1) additional space.

Worst Case: O(log(N))

Best Case: Ω(1)

Average Case: θ(log(N))

Worst Case(Space): O(1)

Program: Implementation of Linear Search algorithm with a set of data in **Python.**

```
from time import time
from random import randrange, randint
import matplotlib.pyplot as plt
from numpy import linspace

MAX, size = int(1e6), int(1e3)
elements, times = [], []
while(size < MAX):
    size *= 2
    arr = [0] * size
    for i in range(size):
        arr[i] = randrange(0, size)
    to_search = randint(0, size * 2)
    start = time()
    # Linear Search
    found = False
    for ele in arr:
        if ele == to_search:
            found = True
            break
    result = "Found!" if found else "Not Found!"
    end = time()
    total_time  = (end - start) * 1000
    elements.append(size)
    times.append(total_time)
    print(f"{result} Time taken to search {to_search} in
array of"
          + f" {size:.0E} numbers is {total_time:.3f}
Milliseconds")

x = linspace(0, max(elements), 100)
y = x/10000
plt.xlabel('Number Count(N)')
plt.ylabel('Time(in Milliseconds)')
plt.plot(x, y, label ='N', color = 'orange')
plt.plot(elements, times, label ='Linear Search', color =
'blue')
plt.grid()
plt.legend()
plt.show()
```

Program: Implementation of Binary Search algorithm with a set of data in **Python.**

```
from time import time
from random import randint
import matplotlib.pyplot as plt
from numpy import linspace
from math import log2

MAX, size = int(1e8), int(1e2)
elements, times = [], []
while(size < MAX):
    size *= 10
    arr = linspace(0, size, num=size, dtype=int)
    to_search = randint(0, size * 4)
    start = time()
    # Binary Search
    found = False
    low, mid, high = 0, 0, len(arr)-1
    while low <= high:
        mid = (low + high)//2
        if arr[mid] < to_search:
            low = mid + 1
        elif arr[mid] > to_search:
            high = mid - 1
        else:
            found = True
            break
    end = time()
    result = "Found!" if found else "Not Found!"
    total_time  = (end - start) * 1000
    elements.append(size)
    times.append(total_time)
    print(f"{result} Time taken to search
{to_search} in array of {size:.0E} numbers is
{total_time:.3f} Milliseconds")
```

Analysis:

Program executed for various sizes of input for Linear Search.

Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	8000	1.134
2	16000	3.607
3	32000	8.112
4	64000	13.125
5	128000	18.748
6	256000	28.123
7	512000	33.752
8	1024000	49.370

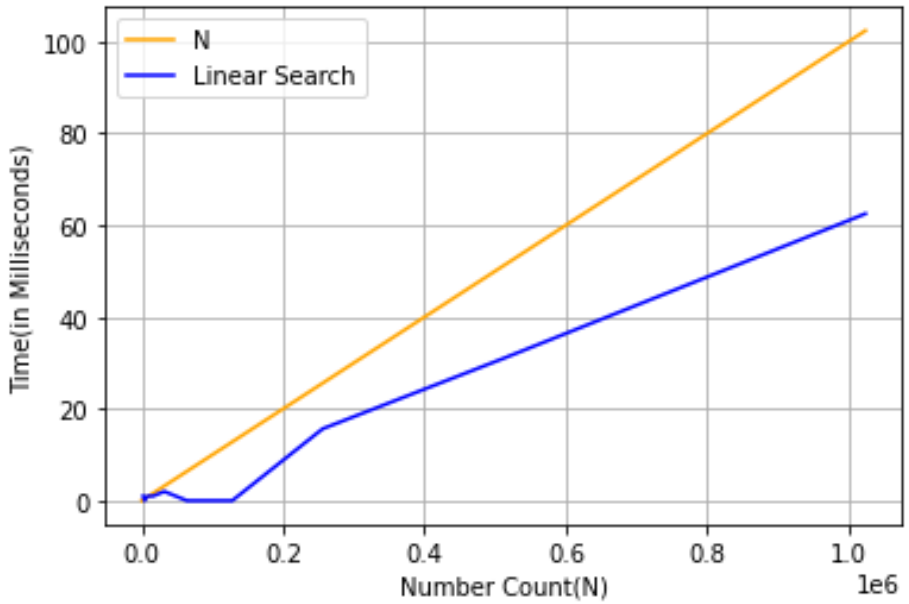
Analysis:

Program executed for various sizes of input for Binary Search.

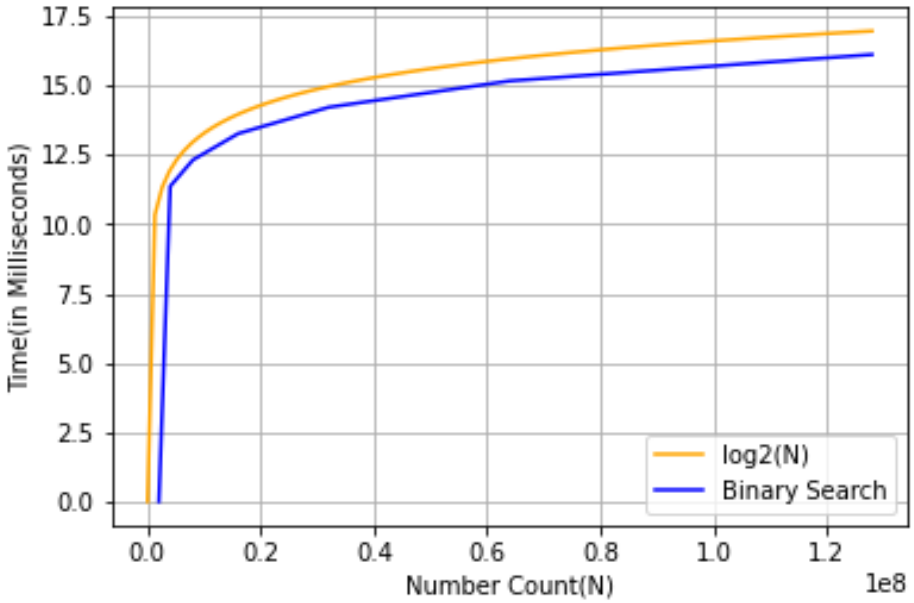
Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	10	0.00
2	100	0.00
3	1000	1.103
4	10000	3.134
5	100000	7.164
6	1000000	12.536
7	10000000	14.254
8	100000000	15.863

Conclusion:

Linear Search's and Binary Search's worst case time Complexity can be visualized by the following graphs.



Maximum Time Taken by **Linear Search** is bounded by O(N).



Maximum Time Taken by **Binary Search** is bounded by O(log(N)).

Practical 4

Aim: Implement Dijkstra Algorithm to find shortest paths to other vertices.

Theory:

Given a graph and a source vertex in the graph, find shortest paths from source to all vertices in the given graph.

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we

find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Djikstra's algorithm (named after its discover, E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.

For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Dijkstra's algorithm is the

best known algorithm for the single source shortest paths problem. This algorithm is applicable to graphs with nonnegative weights only and finds the shortest paths to a graph's vertices in order of their distance from a given source. It finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. It is applicable to both undirected and directed graphs The algorithm generates a SPT (shortest path tree) with given source as root and maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Algorithm(Dijkstra's):

```
function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:
 6          dist[v] ← INFINITY
 7          prev[v] ← UNDEFINED
 8          add v to Q
 9      dist[source] ← 0
10
11      while Q is not empty:
12          u ← vertex in Q with min dist[u]
13
14          remove u from Q
15
16          for each neighbor v of u:           // only v
that are still in Q
17              alt ← dist[u] + length(u, v)
18              if alt < dist[v]:
19                  dist[v] ← alt
20                  prev[v] ← u
21
22      return dist[], prev[]
```

Complexity of Dijkstra's Algorithm:

The Time efficiency for graphs represented by their weight matrix and the priority queue implemented as an unordered array and for graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is **O(|E|log|V|)**. Our implementation has Time Complexity of **O(V^2)**.

Program: Implementation of algorithm with a set of data in **Python.**

```
from time import time
from random import randrange
import matplotlib.pyplot as plt
import numpy as np
from sys import maxsize as MAX_INT

class Graph():
    def __init__(self, vertices, rand=True):
        self.rand = rand
        self.V = vertices
        self.graph = [[0 for column in
range(vertices)]
                      for row in range(vertices)]
        if rand:
            sz = self.V
            for i in range(int(sz**2)):
                row = randrange(0, sz)
                col = randrange(0, sz)
                self.graph[row][col] = randrange(0,
sz * i+1)

    def minDistance(self, dist, sptSet):
        min = MAX_INT
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index

    def printSolution(self, dist):
        print("Vertex \tDistance from Source")
        for node in range(self.V):
            print(node, "\t", dist[node])
```

```
    def dijkstra(self, src):

        dist = [MAX_INT] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):
            u = self.minDistance(dist, sptSet)
            sptSet[u] = True
            for v in range(self.V):
                if self.graph[u][v] > 0 and \
                sptSet[v] == False and \
                dist[v] > dist[u] +
self.graph[u][v]:
                            dist[v] = dist[u] +
self.graph[u][v]
        if not self.rand:
            self.printSolution(dist)


MAX, size = int(625), 25
elements, times = [0], [0]
print("Dijkstra's Algorithm took:")
while(size <= MAX):
    size += 25
    g = Graph(size)
    start = time()
    # Dijkstra
    g.dijkstra(0)
    end = time()
    tt  = (end - start) * 1000
    elements.append(size)
    times.append(tt)
    print(f"{tt:.3f}ms for Graph with {size}
vertices ")
```

For Plotting Graph:

```
x = np.linspace(0, max(elements), 100)
y = (x/30) ** 2
plt.xlabel('Number Count(N)')
plt.ylabel('Time(in Milliseconds)')
plt.plot(x, y, label ='V^2', color =
'orange')
plt.plot(elements, times, label
='Dijkstra\'s', color = 'blue')
plt.grid()
plt.legend()
plt.show()
```

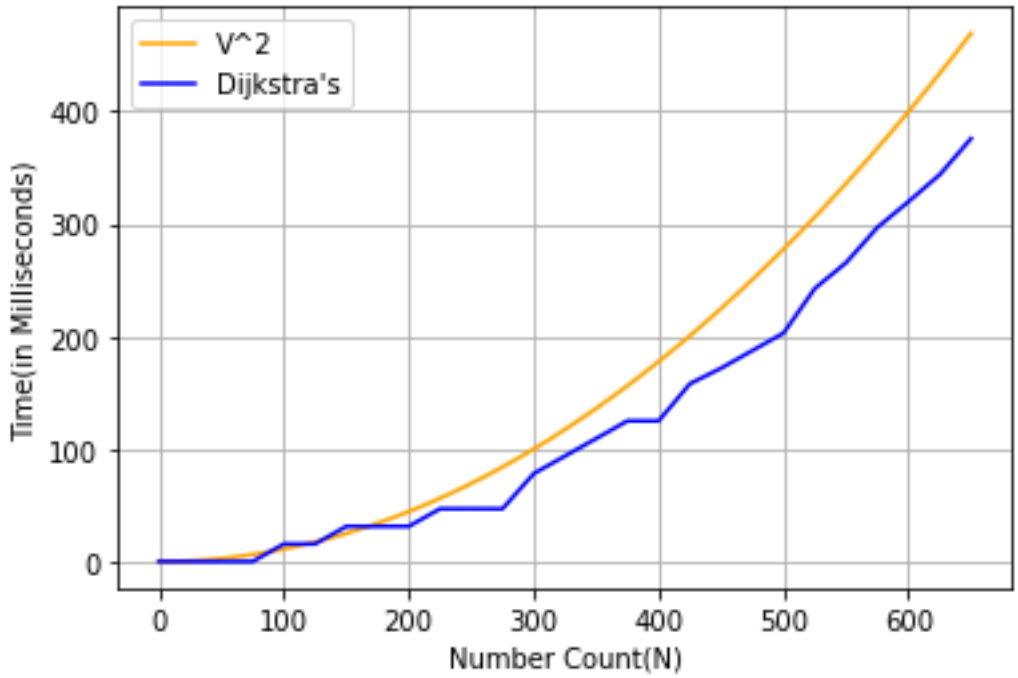
Analysis:

Program to be executed for various sizes of input.

Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	50	0.000
2	75	2.501
3	100	15.625
4	125	15.630
5	150	31.249
6	175	31.251
7	200	31.244
8	225	46.874
9	250	46.896
10	275	46.866
11	300	78.123
12	325	93.750
13	350	109.354
14	375	125.025
15	400	124.990
16	425	157.989
17	450	171.885
18	475	187.503
19	500	203.116
20	525	242.773
21	550	265.620
22	575	296.901
23	600	319.941
24	625	343.941
25	650	375.976

Conclusion:

Dijkstra's Algorithm's worst case time Complexity can be visualized by the following graphs.



Maximum Time Taken by Dijkstra's Algorithm is bounded by **O(V^2)** where V is number of vertices.

Practical 5

Aim: Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm and analyse it's complexity.

Theory:

Given an undirected and connected graph G = (V, E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G). The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are: Cluster Analysis, Handwriting recognition, Image segmentation. Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an

algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an algorithm to detect cycle. This is another method based on Union-Find. This method assumes that the graph doesn’t contain any self-loops.

We can keep track of the subsets in a 1D array, let’s call it parent[].

Algorithm(Kruskal's):

```
KRUSKAL(G):
Step 1. A = ∅
Step 2. For each vertex v ∈ G.V:
Step 3. MAKE-SET(v)
Step 4. For each edge (u, v)∈
G.E ordered by increasing order
by weight(u,v)
Step 5. if FIND-SET(u) ≠ FIND-
SET(v):
Step 6. A = A ∪ {(u, v)}
Step 7. UNION(u, v)
Step 8. return A
```

Complexity of Kruskal's Algorithm:

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be **O(|E|log|V|)**, which is the overall Time Complexity of the algorithm.

Program: Implementation of algorithm with a set of data in **Python.**

```
from time import time
from random import randrange
import matplotlib.pyplot as plt
import numpy as np
from sys import maxsize as MAX_INT

class Graph():
    def __init__(self, vertices, rand=True):
        self.rand = rand
        self.costs = []
        self.mincost = 0
        self.V = vertices
        self.parent = [i for i in range(self.V)]
        self.graph = [[MAX_INT for column in
range(vertices)]
                      for row in range(vertices)]
        if rand:
            sz = self.V
            for i in range(int(sz**2)):
                row = randrange(0, sz)
                col = randrange(0, sz)
                self.graph[row][col] =
randrange(0, sz * i+1)

    def find(self, i):
        while self.parent[i] != i:
            i = self.parent[i]
        return i

    def union(self, i, j):
        a = self.find(i)
        b = self.find(j)
        self.parent[a] = b
```

```
    def kruskalMST(self):
        for i in range(self.V):
            self.parent[i] = i
        edge_count = 0
        while edge_count < self.V - 1:
            min_c = MAX_INT
            a = -1
            b = -1
            for i in range(self.V):
                for j in range(self.V):
                    if self.find(i) !=
self.find(j) and \
                    self.graph[i][j] < min_c:
                        min_c = self.graph[i][j]
                        a = i
                        b = j
            self.union(a, b)
            self.costs.append([edge_count, a, b,
min_c])
            edge_count += 1
            self.mincost += min_c

        if not self.rand:
            self.printSolution()

    def printSolution(self):
        for i in range(len(self.costs)):
            edge_count, min_c = self.costs[i][0],
self.costs[i][3]
            a, b = self.costs[i][1],
self.costs[i][2]
            print(f'Edge {edge_count}:({a}, {b})
cost:{min_c}')
        print(f"Minimum cost = {self.mincost}")
```

```
MAX, size = int(90), 15
elements, times = [0], [0]
print("Kruskal's Algorithm took:")
while(size < MAX):
    size += 3
    g = Graph(size)
    start = time()
    # Kruskal's
    g.kruskalMST()
    end = time()
    tt  = (end - start) * 1000
    elements.append(size)
    times.append(tt)
    print(f"{tt:.3f}ms for Graph with
{size} vertices ")

# Code for plotting Graph
x = np.linspace(0, max(elements), 100)
y = [0] * 100
for i in range(1, len(x)):
    y[i] = (x[i]/0.25 * (log2(x[i]/0.25)))
plt.xlabel('Number Count(N)')
plt.ylabel('Time(in Milliseconds)')
plt.plot(x, y, label ='Elog(V)', color =
'orange')
plt.plot(elements, times, label
='Kruskal\'s', color = 'blue')
plt.grid()
plt.legend()
plt.show()
```

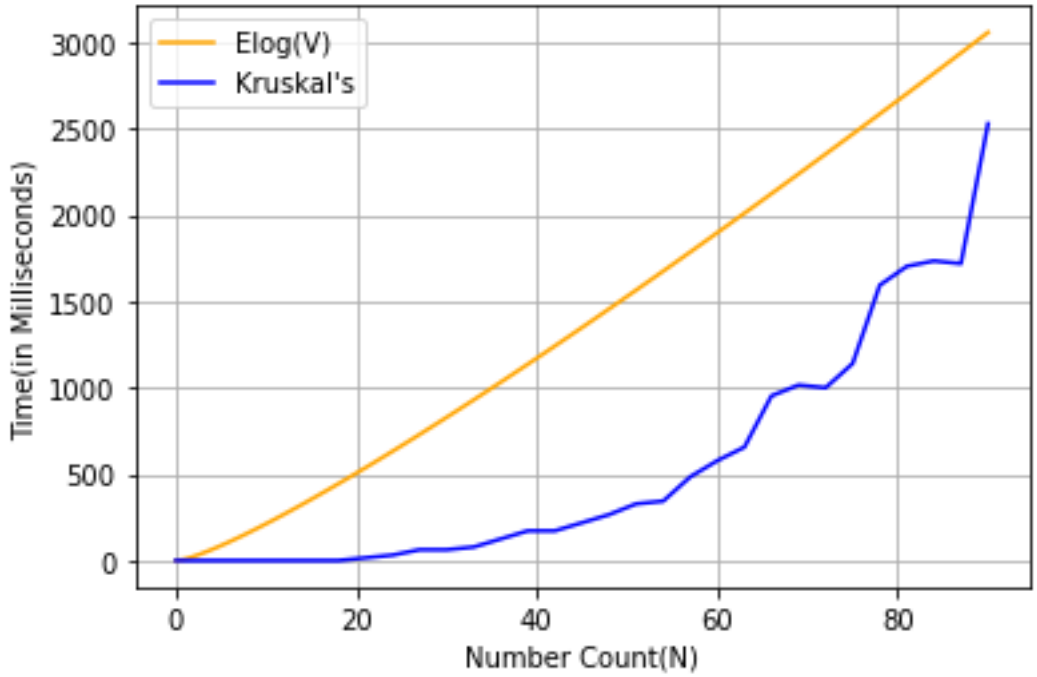
Analysis:

Program to be executed for various sizes of input.

Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	18	15.601
2	21	15.621
3	24	46.875
4	27	46.872
5	30	78.130
6	33	78.125
7	36	109.377
8	39	156.254
9	42	156.267
10	45	203.095
11	48	234.393
12	51	359.350
13	54	374.998
14	57	421.875
15	60	546.876
16	63	650.976
17	66	796.871
18	69	828.147
19	72	1015.627
20	75	1446.299
21	78	1554.588
22	81	1625.024
23	84	1937.477
24	87	2671.871
25	90	3001.726

Conclusion:

Kruskal's Algorithm's worst case time Complexity can be visualized by the following graphs.



Maximum Time Taken by Kruskal's algorithm is bounded by **Elog(V)** where E is number of Edges and V is number of Vertices.

Practical 6

Aim: Implement the All pair shortest path Algorithm using dynamic programming and analyze its complexity

Theory:

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warhshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

This algorithm follows the dynamic programming approach to find the shortest paths.

Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A0 of dimension v*v where v is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

   Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

2. Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

   Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).

   That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then

the cell is filled with A[i][k] + A[k][j].

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.

For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.

3. Similarly, A2 is created using A1. The elements in the second column and the second row are left as they are.

   In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.

4. Similarly, A3 and A4 is also created.

5. A4 gives the shortest path between each pair of vertices.

Algorithm (Floyd Warshall):

```
let dist be a |V| × |V| array of minimum distances
initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)// The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

Complexity of Floyd Warshall Algorithm:

The Floyd-Warshall algorithm is a graph-analysis algorithm that calculates shortest paths between all pairs of nodes in a graph. It is a **dynamic programming algorithm** with $O(|V|^3)$ time complexity and $O(|V|^2)$ space complexity.

The Floyd-Warshall all-pairs shortest path runs in $O(V^3)$ time, which is asymptotically no better than n calls to Dijkstra's algorithm. However, the loops are so tight and the program so short that it runs better in practice.

Program: Implementation of algorithm with a set of data in **Python**.

```
from time import time
from random import randrange
import matplotlib.pyplot as plt
import numpy as np
from sys import maxsize as MAX_INT

class Graph():
    def __init__(self, vertices, rand=True):
        self.rand = rand
        self.V = vertices
        self.graph = [[MAX_INT for column in range(vertices)]
                      for row in range(vertices)]
        if rand:
            sz = self.V
            for i in range(int(sz**1.5)):
                row = randrange(0, sz)
                col = randrange(0, sz)
                self.graph[row][col] = randrange(0, sz * i+1)


    def floydWarshall(self):
        dist = list(map(lambda i: list(map(lambda j: j, i)),
self.graph))
        for k in range(self.V):
            for i in range(self.V):
                for j in range(self.V):
                    dist[i][j] = min(dist[i][j], dist[i][k] +
dist[k][j])
        if not self.rand:
            self.printSolution(dist)


    def printSolution(self, dist):
        print("All Pairs Shortest Path Matrix")
        for i in range(self.V):
            for j in range(self.V):
                if(dist[i][j] == MAX_INT):
                    print("%7s" % ("INF"), end='')
                else:
                    print("%7d\t" % (dist[i][j]), end='')
                if j == self.V-1:
                    print("")
```

```
MAX, size = int(90), 15
elements, times = [0], [0]
print("Floyd Warshall Algorithm took:")
while(size < MAX):
    size += 3
    g = Graph(size)
    start = time()
    # Floyd Warshall
    g.floydWarshall()
    end = time()
    tt  = (end - start) * 1000
    elements.append(size)
    times.append(tt)
    print(f"{tt:.3f}ms for Graph with
{size} vertices ")

# Code for plotting Graph
x = np.linspace(0, max(elements), 100)
y = (x/11) ** 3
plt.xlabel('Number of Vertices(V)')
plt.ylabel('Time(in Milliseconds)')
plt.plot(x, y, label ='V^3', color =
'orange')
plt.plot(elements, times, label ='Floyd
Warshall', color = 'blue')
plt.grid()
plt.legend()
plt.show()
```

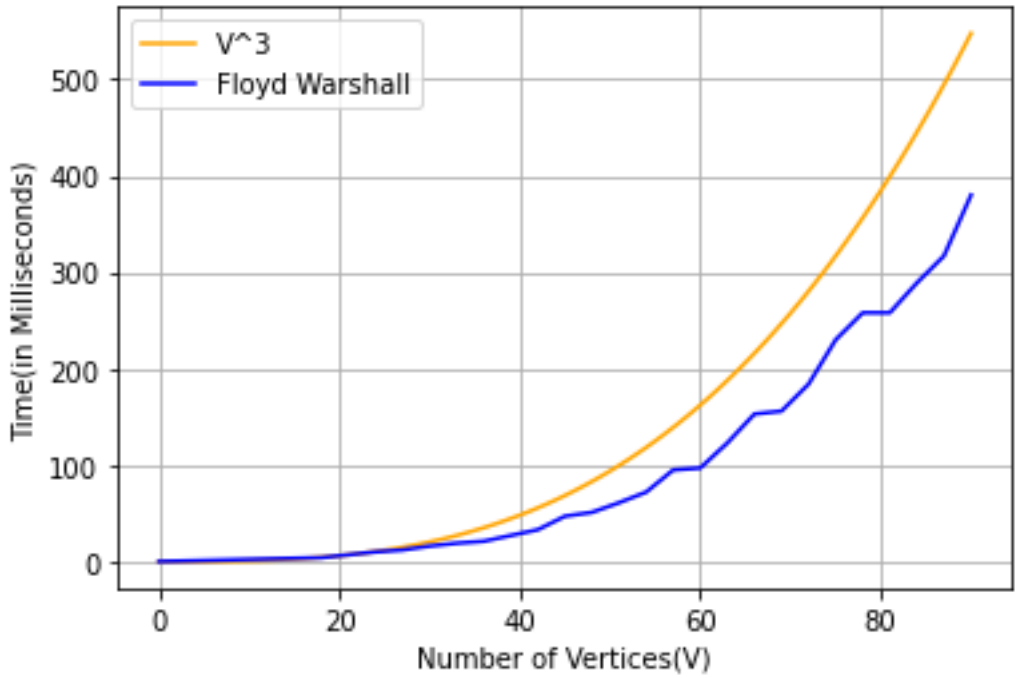
Analysis:

Program executed for various sizes of input.

Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	18	4.999
2	21	6.996
3	24	9.000
4	27	14.017
5	30	16.516
6	33	21.000
7	36	25.003
8	39	27.012
9	42	33.007
10	45	42.996
11	48	51.000
12	51	67.990
13	54	72.994
14	57	95.000
15	60	100.998
16	63	132.983
17	66	156.025
18	69	159.991
19	72	209.999
20	75	201.999
21	78	236.974
22	81	262.031
23	84	316.001
24	87	320.996
25	90	391.969

Conclusion:

Floyd Warshall Algorithm's worst case time Complexity can be visualized by the following graphs.



Maximum Time Taken by Floyd Warshall's algorithm is bounded by $|V|^3$ where V is number of Vertices.

Practical 7

Aim: Implement the 4-queens problem using backtracking technique and analyse its complexity.

Theory:

Backtracking is similar to the brute force approach where it tries all of the solutions but the only difference is that it eliminates/avoids the partial candidate solutions as soon as it finds that that path cannot lead to a solution.

The 4-Queens Problem consists in placing four queens on a 4 x 4 chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column or the same diagonal.

The following figure illustrates a solution to the 4-Queens Problem: none of the 4 queens can capture each other.

Although this particular problem isn’t very impressive, keep in mind that you can generalize it to n x n chessboards with n >= 4.

The algorithm to solve this problem uses backtracking, but we will unroll the recursion. The basic idea is to place queens column by column, starting at the left.

New queens must not be attacked by the ones to the left that have already been placed on the board. We place another queen in the next column if a consistent position is found. All rows in the current column are checked. We have found a solution if we placed a queen in the rightmost column.

Algorithm(N-Queen Backtracking):

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column.
   Do following for every tried row.
    a) If the queen can be placed safely in this row
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing the queen in [row, column] leads to
       a solution then return true.
    c) If placing queen doesn't lead to a solution then
       unmark this [row, column] (Backtrack) and go to
       step (a) to try other rows.
4) If all rows have been tried and nothing worked,
   return false to trigger backtracking.
```

Complexity of N-Queens Problem's Backtracking Algorithm:

The power of the set of all possible solutions of the n queen's problem is n! and the bounding function takes a linear amount of time to calculate, therefore the running time of the n queens problem is **O(N!)**.

Program: Implementation of algorithm with a set of data in **Python**.

```
from time import time
import matplotlib.pyplot as plt
import numpy as np

class Queen:
    def __init__(self, N=4, rand=True):
        self.N = N
        self.rand = rand
        self.rd = [0]*N*2
        self.ld = [0]*N*2
        self.cl = [0]*N*2

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

    def solveNQUtil(self, board, col):
        if (col >= self.N):
            return True
        for i in range(self.N):
            if ((self.ld[i-col+self.N-1]!=1 and
                self.rd[i+col]!=1) and self.cl[i]!=1):
                board[i][col] = 1
                self.ld[i-col+self.N-1] = self.rd[i+col] =
self.cl[i] = 1
                if (self.solveNQUtil(board, col+1)):
                    return True
                board[i][col] = 0
                self.ld[i-col+self.N-1] = self.rd[i+col] =
self.cl[i] = 0
        return False

    def solveNQ(self):
        board = [[0 for column in range(self.N)]for row in
range(self.N)]
        if (self.solveNQUtil(board, 0) == False):
            print("Solution does not exist")
            return False
        if not self.rand:
            self.printSolution(board)
        return True
```

```
MAX, size = 22, 3
elements, times = [0], [0]
print("N Queen's Problem took:")
while(size < MAX):
    size += 1
    q = Queen(size)
    start = time()
    # N-Queen's
    q.solveNQ()
    end = time()
    tt  = (end - start) * 1000
    elements.append(size)
    times.append(round(tt,3))
    print(f"{tt:.3f}ms for {size}
Queens")


# Code for plotting Graph
x = np.linspace(0, max(elements), 100)
y = 2 ** (x/1.45)
plt.xlabel('Number of Queens(N)')
plt.ylabel('Time(in Milliseconds)')
plt.plot(x, y, label ='2^N', color =
'orange')
plt.plot(elements, times, label ='N-
Queen\'s', color = 'blue')
plt.grid()
plt.legend()
plt.show()
```

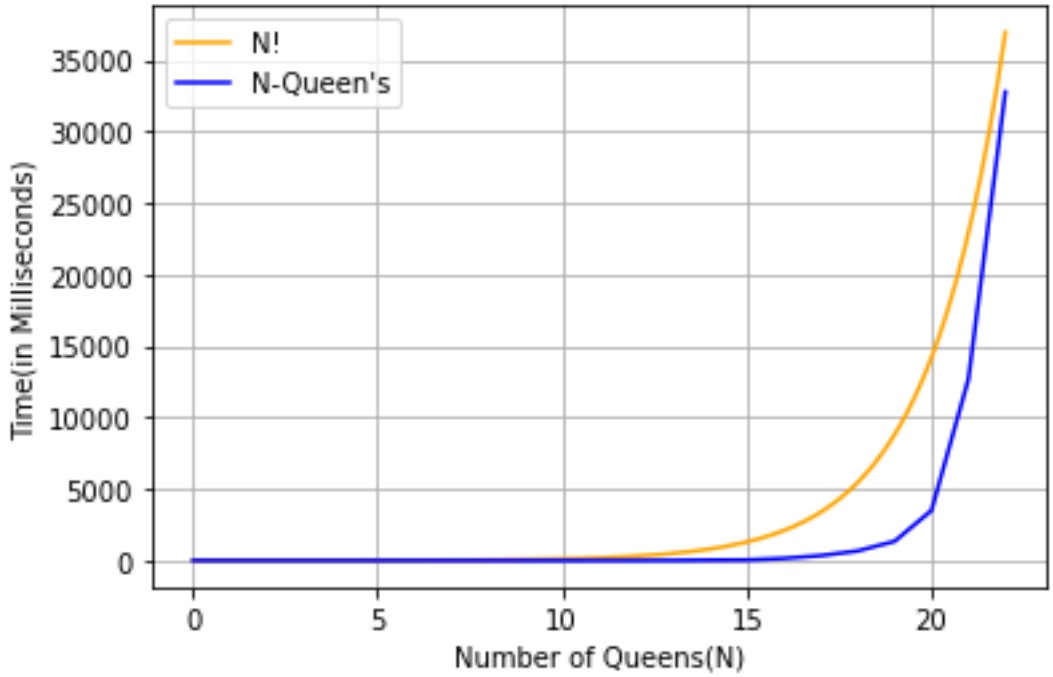
Analysis:

Program to be executed for various sizes of input.

Sr. No.	Input Size(n)	Time Complexity(Milliseconds)
1	4	0.0
2	5	0.0
3	6	0.0
4	7	0.0
5	8	0.001
6	9	0.003
7	10	0.018
8	11	1.0
9	12	1.034
10	13	3.008
11	14	7.010
12	15	26.002
13	16	39.996
14	17	149.028
15	18	342.701
16	19	670.0
17	20	1342.050
18	21	3499.016
19	22	12683.04
20	23	32764.0

Conclusion:

N-Queen Problem's worst case time Complexity can be visualized by the following graphs.



Maximum Time Taken by N-Queen's Backtracking algorithm is bounded by **N!** where N is number of Queen's to place on the an **N X N** Chessboard.