GOVERNMENT COLLEGE OF ENGINEERING SENGIPETTI, THANJAVUR

NAME :

YEAR :

BRANCH :

REGISTER NUMBER :

Certified to be the Bonafide record of work done by the above student in ***FOURTH SEMESTER*** *in* ***CS3401-ALGORITHMS LABORATORY*** *during the year 2024-2025.*

Signature of the staff in-charge *Signature of the Head of the Department*

Submitted for the practical examination held on________________.

Internal Examiner *External Examiner*

INDEX

14.	15/04/25	Implementation of Floyd's Algorithm for All – Pairs Shortest Paths	70		
15.	15/04/25	Implementation of Transitive Closure of directed graph using Warshall's Algorithm	75		
16.	22/04/25	Implementation of MinMax problem using Divide and Conquer Technique	80		
17.	26/04/25	Implementation of Merge Sort	85		
18.	26/04/25	Implementation of Quick Sort	91		
19.	29/04/25	Implementation of N-Queens problem using Backtracking	96		
20.	06/05/25	Implementation of Travelling Salesperson problem using Approximation Algorithm	101		
21.	13/05/25	Implementation of Randomized Algorithm for finding the k th Smallest Number	107		

EX.NO:1 IMPLEMENTATION OF LINEAR SEARCH

25/02/25

AIM:

To implement linear search to determine the time required to search for an element. Repeat the experiment for different values of n, the no of elements in the list to be searched and plot a graph of the time taken versus n.

PSEUDOCODE:

PROGRAM:

```
import time

import matplotlib.pyplot as plt

def linear_search(arr, target):

  nbasicop=0

  for index in range(len(arr)):

    nbasicop+=1

    if arr[index] == target:

      return index,nbasicop

  return -1,nbasicop

def measure_time(n,basicops):

  A=[i for i in range(n)]

  start=time.time()

  index,nbasicop=linear_search(A,n)

  end=time.time()

  basicops.append(nbasicop)

  return end-start

n_values=[10,100,1000,10000,100000,1000000]

basicops=[]

times=[measure_time(n,basicops) for n in n_values]

print(n_values)

print(basicops)

print(times)

plt.plot(n_values,basicops,label="basicop",marker="o")

plt.title("Performance of Linear search in terms of number of Basic Operation")
```

```
plt.xlabel("N values")

plt.ylabel("Basicops")

n=[i for i in range(10,1000000)]

plt.plot(n,n,label="O(n) Plot")

plt.legend()plt.plot(n_values,times)

plt.xlabel("N values")

plt.ylabel("Time taken(seconds)")

plt.title("Time Complexity of Linear Search")

plt.show()
```

OUTPUT:

N Values:

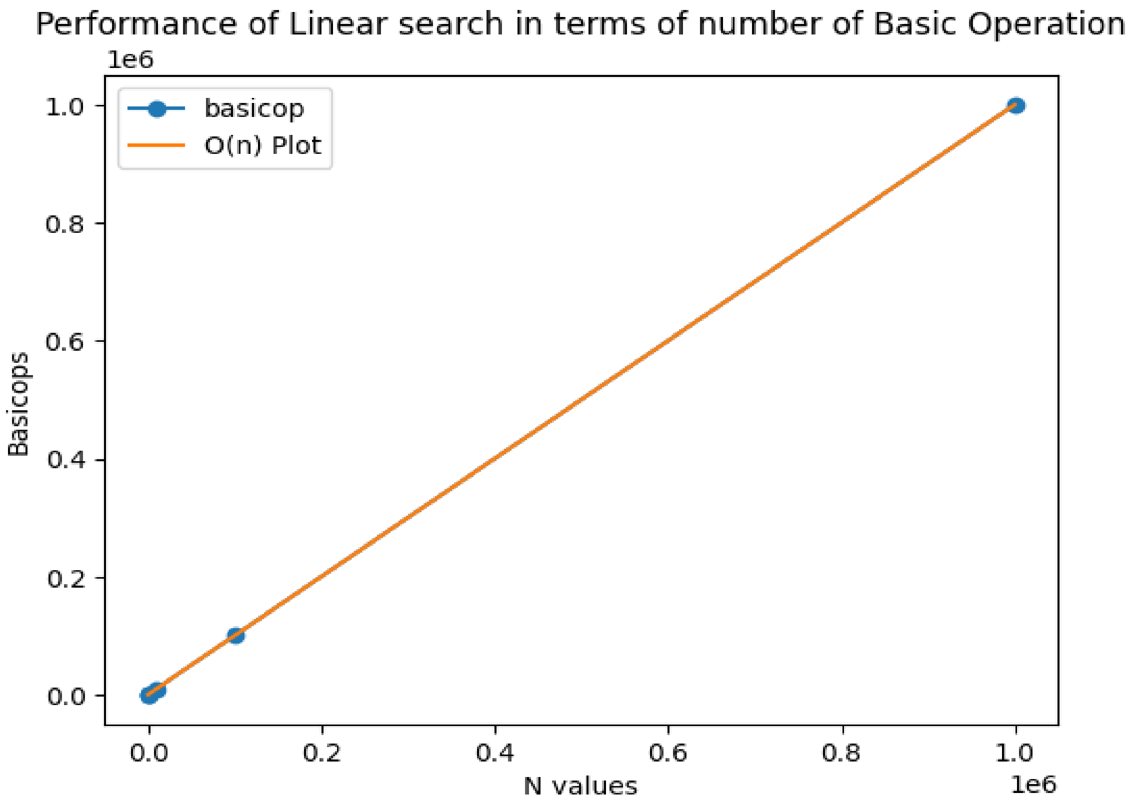
[10, 100, 1000, 10000, 100000, 1000000]

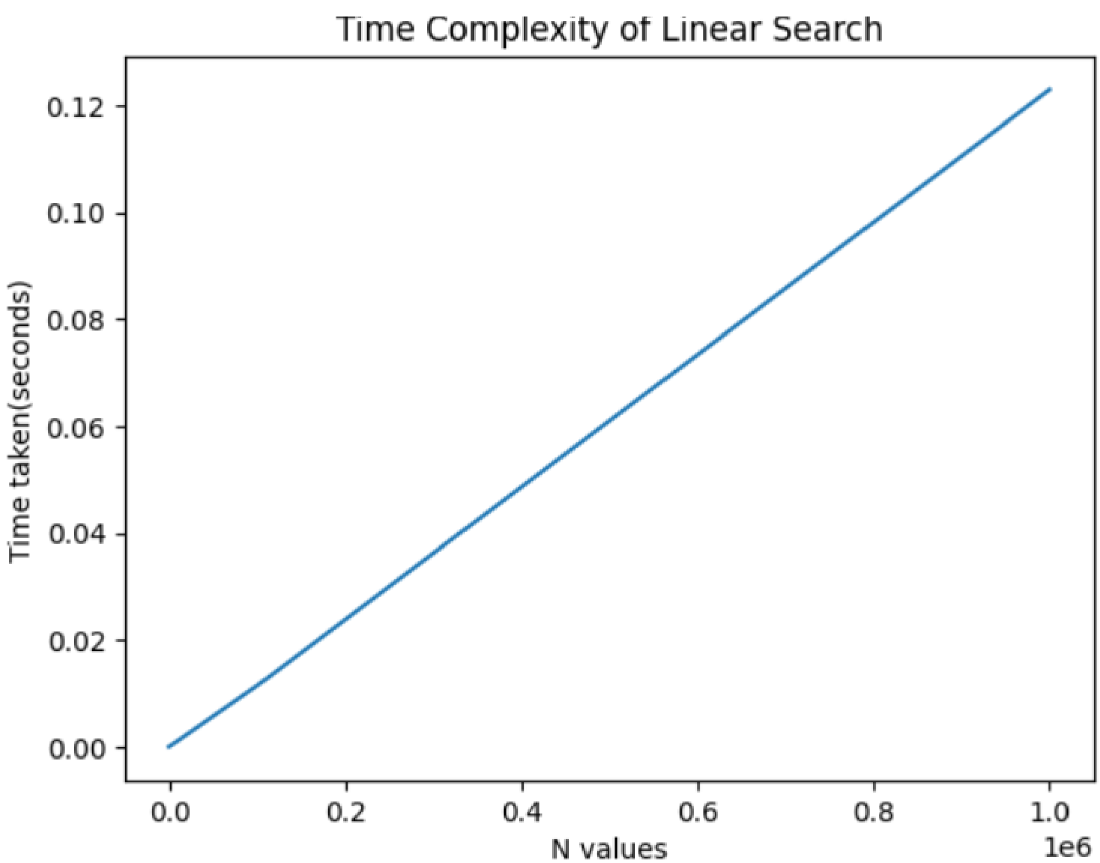
Basic Operation:

[10, 100, 1000, 10000, 100000, 1000000]

Times:

[5.9604644775390625e-06, 7.62939453125e-06, 7.867813110351562e-05, 0.0009257793426513672, 0.006898403167724609, 0.062090158462524414]


Performance of Linear search in terms of number of Basic Operation
1e6
1.0
0.8
0.6
0.4
0.2
0.0
Basicops
basicop
O(n) Plot
0.0
0.2
0.4
0.6
0.8
1.0
N values
1e6


Time Complexity of Linear Search
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Time taken(seconds)
0.0
0.2
0.4
0.6
0.8
1.0
N values
1e6

RESULT:

Thus the implementation of linear search to determine the time required to search a element for the different value of n , the number of element in the list to be searched and plot a graph of the time taken versus n has been executed and verified successfully.

EX.NO:2 IMPLEMENTATION OF ITERATIVE BINARY SEARCH

25/02/25

AIM:

To implement Iterative Binary Search to determine the time required to search for an element. Repeat the experiment for different values of n, the no of elements in the list to be searched and plot a graph of the time taken versus n.

PSEUDOCODE:

PROGRAM:

```
import time

import matplotlib.pyplot as plt

def binary_search(arr, low, high,target, basicops):

  low = 0

  high = len(arr) - 1

  while low <high:

    mid = (low + high) // 2

    basicops+=1

    if arr[mid] == target:

      return mid

    elif arr[mid] < target:

      low = mid + 1

    else:

      high = mid - 1

  return -1,basicops

def measure_time1(n,basicops):

 basicop=0

 A=[i for i in range(n)]

 start=time.time()

 index,nbasicop=binary_search(A,0,n-1,n+1,basicop)

 end=time.time()

 basicops.append(nbasicop)

 return end-start

n_values=[10,100,1000,10000,100000,1000000]
```

```
basicops=[]

times=[measure_time1(n,basicops) for n in n_values]

print("N Values:")

print(n_values)

print("Basic Operation:")

print(basicops)

print("Time Taken:")

print(times)

plt.plot(n_values,times,label="Times")

plt.title("Time Taken Vs N_Values")

plt.xlabel("N values")

import math

logvalue=[math.log2(i) for i in n_values]

plt.plot(n_values,basicops,label='Basicop')

plt.plot(n_values,logvalue,label='lgn plot')

plt.xlabel("n values")

plt.title("Performance of Iterative Binary Search in terms of number of basicop")

plt.ylabel(" basic operation")

plt.legend()

plt.show()
```

OUTPUT:

N Values:

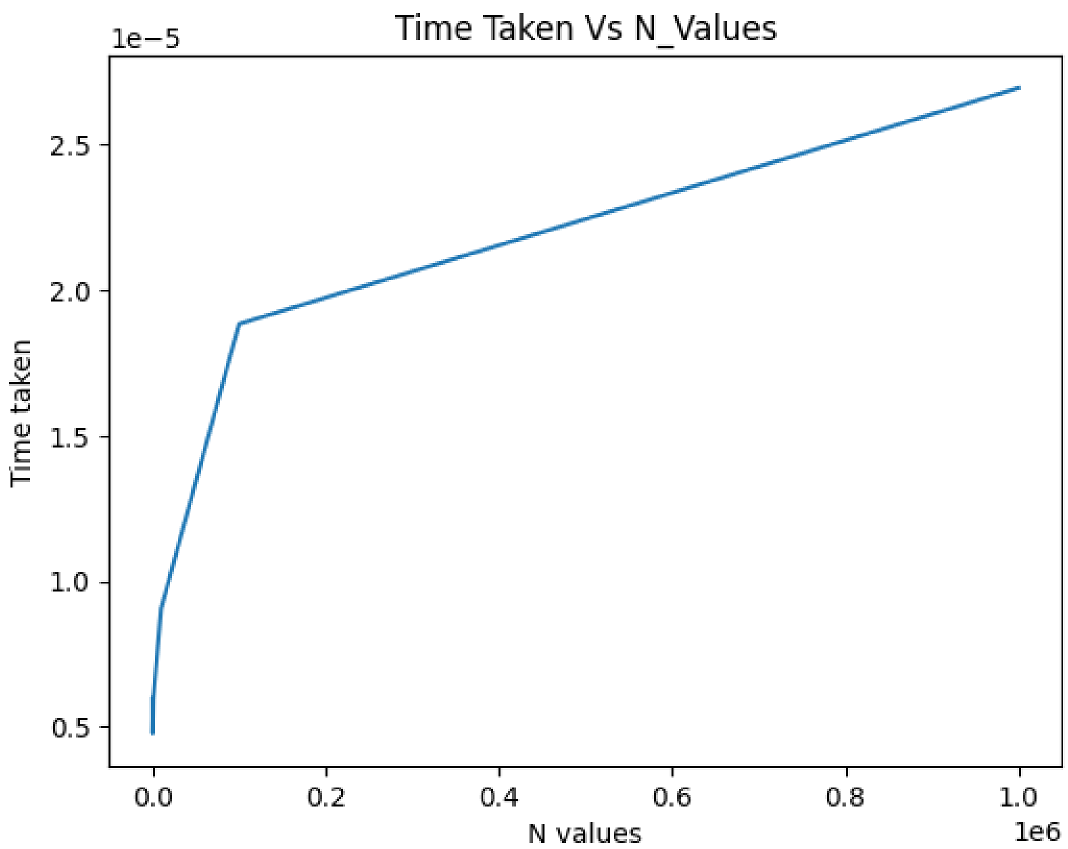
[10, 100, 1000, 10000, 100000, 1000000]

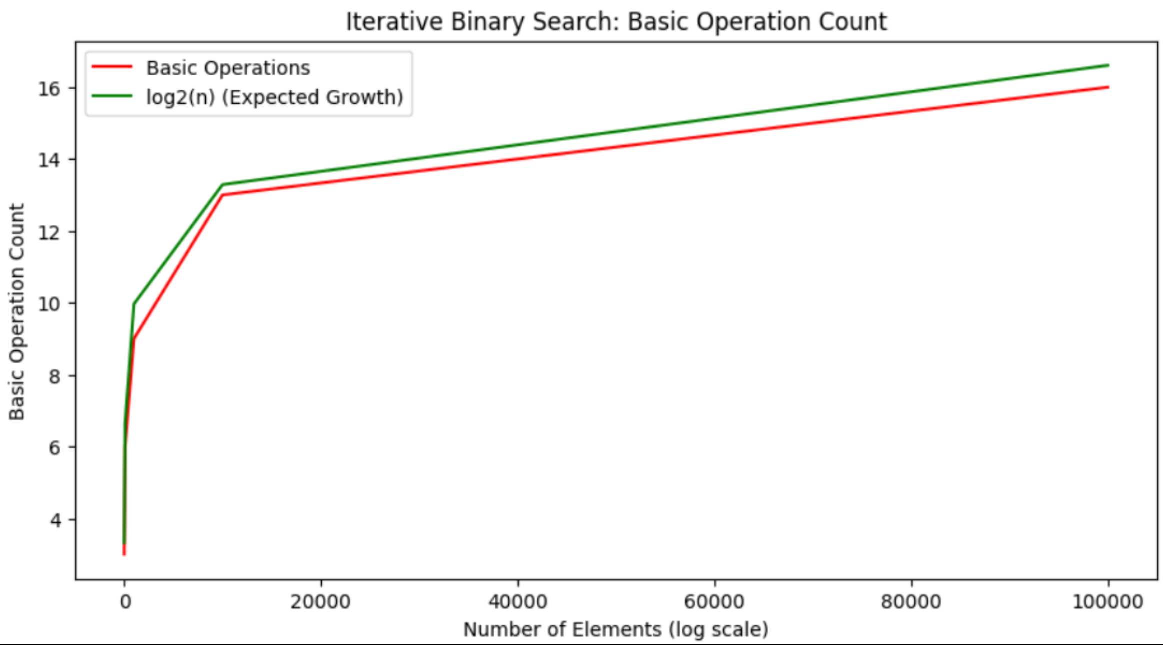
Basic Operation:

[4, 7, 10, 14, 17, 20]

Time Taken:

[5.9604644775390625e-06, 4.76837158203125e-06, 5.9604644775390625e-06, 9.059906005859375e-06, 1.8835067749023438e-05, 2.6941299438476562e-05]





RESULT:

Thus the implementation of Iterative Binary Search to determine the time required to search a element for the different value of n , the number of element in the list to be searched and plot a graph of the time taken versus n has been executed and verified successfully.

EX.NO:3 ***IMPLEMENTATION OF RECURSIVE BINARY SEARCH***

04/03/25

AIM:

To implement recursive binary search to determine the time required to search for an element. Repeat the experiment for different values of n, the no of elements in the list to be searched and plot a graph of the time taken versus n.

PSEUDOCODE:

PROGRAM:

```
import time

import matplotlib.pyplot as plt

def binary_search(arr, low, high, x,nbasicop):

   if high >= low:

     nbasicop=nbasicop+1

     mid = (high + low) // 2

     if arr[mid] == x:

       return mid,nbasicop

     elif arr[mid] > x:

      return binary_search(arr, low, mid - 1, x,nbasicop)

     else:

      return binary_search(arr, mid + 1, high, x,nbasicop)

    else:

     return -1,nbasicop

def measure_time(n,basicop):

    a=[i for i in range(n)]

    nbasicop=0

    start=time.time()

    index,nbasicops=binary_search(a,0,len(a)-1,a[0],nbasicop)

    end=time.time()

    basicop.append(nbasicops)

    return end-start

import math

nv=[10,100, 1000, 10000, 100000, 1000000]
```

```
basicop=[]

time=[measure_time(n,basicop)for n in nv]

logvalue=[math.log2(n) for n in nv]

print(nv)

print(basicop)

print(time)

print(logvalue)

plt.plot(nv,time)

plt.xlabel("N values")

plt.ylabel("Time taken")

plt.plot(nv,basicop,label='Basicop')

plt.plot(nv,logvalue,label='Log')

plt.xlabel("n values")

plt.ylabel(" basic operation")

plt.title("Performance of Recursive Binary Search in terms of number of basic operation")

plt.legend()

plt.show()
```

OUTPUT:

N Sizes:

[10, 100, 1000, 10000, 100000, 1000000]

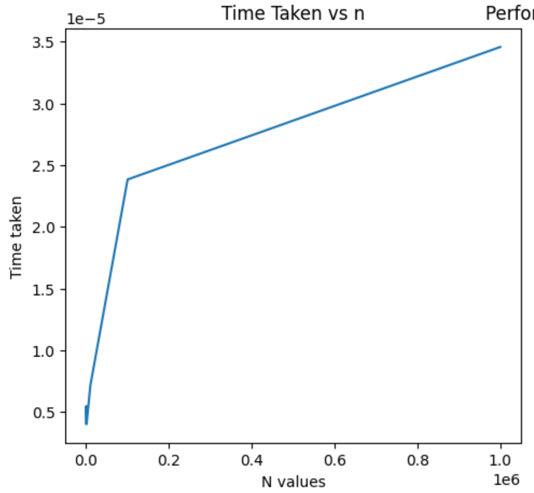
Basic Operation:

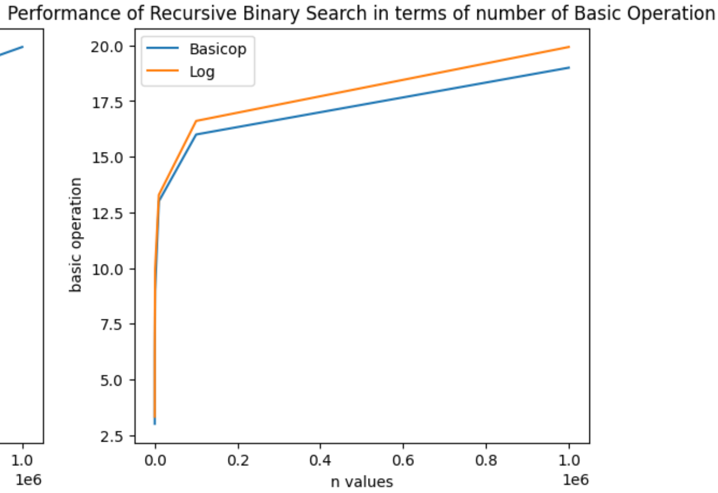
[3, 6, 9, 13, 16, 19]

Times:

[4.0531158447265625e-06, 5.4836273193359375e-06, 4.0531158447265625e-06, 7.152557373046875e-06, 2.384185791015625e-05, 3.457069396972656e-05]

[3.321928094887362, 6.643856189774724, 9.965784284662087, 13.287712379549449, 16.609640474436812, 19.931568569324174]





RESULT:

Thus the implementation of recursive binary search to determine the time required to search a element for the different value of n ,the number of element in the list to be searched and plot a graph of the time taken versus n has been executed and verified successfully.

EX.NO:4 IMPLEMENTATION OF INTERPOLATION SEARCH

04/03/25

AIM:

To implement interpolation search to determine the time required to search for an element. Repeat the experiment for different values of n, the no of elements in the list to be searched and plot a graph of the time taken versus n.

PSEUDOCODE:

PROGRAM:

```
import time

import matplotlib.pyplot as plt

def interpolate_search(arr, low, high, target,nbasicop):

    if high>=low and arr[low]<=arr[high]:

      nbasicop+=1

      pos= low+((high-low)*(target-arr[low]) // (arr[high]-arr[low]))

      if arr[pos] == target:

        return pos,nbasicop

      elif arr[pos] < target:

        return interpolate_search(arr, pos + 1,high, target,nbasicop)

      else:

        return interpolate_search(arr, low,pos - 1, target,nbasicop)

     return -1,nbasicop

def measure_time(n,basicop):

    nbasicop=0

    a=[i for i in range(n)]

    start=time.time()

    x=len(a)-1

    index,nbasicops=interpolate_search(a,0,len(a)-1,x,nbasicop)

    end=time.time()

    basicop.append(nbasicops)

    return end-start

import math

nv=[10,100, 1000, 10000, 100000, 1000000]
```

```
basicop=[]

time=[measure_time(n,basicop)for n in nv]

logvalue=[math.log2(n) for n in nv]

print(nv)

print(basicop)

print(time)

print(logvalue)

plt.plot(nv,time)

plt.xlabel("N values")

plt.ylabel("Time taken")

plt.plot(nv,basicop,label='Basicop')

plt.plot(nv,logvalue,label='Log')

plt.xlabel("n values")

plt.ylabel(" basic operation")

plt.title("Performance of Interpolation Search in terms of number of basic operation")

plt.legend()

plt.show()
```

<u>OUTPUT:</u>

N Values

[10, 100, 1000, 10000, 100000, 1000000]

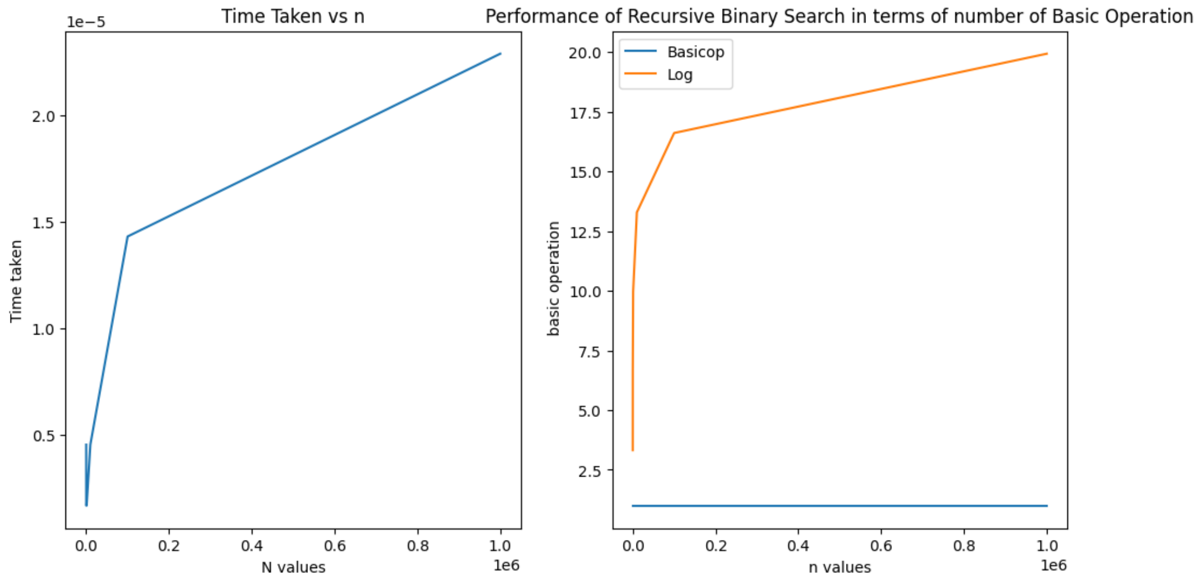
Basic Operation

[1, 1, 1, 1, 1, 1]

times:

[4.5299530029296875e-06, 1.6689300537109375e-06, 1.6689300537109375e-06, 4.5299530029296875e-06, 1.430511474609375e-05, 2.288818359375e-05]

[3.321928094887362, 6.643856189774724, 9.965784284662087, 13.287712379549449, 16.609640474436812, 19.931568569324174]



RESULT:

Thus the implementation of interpolation search to determine the time required to search a element for the different value of n, the number of element in the list to be searched and plot a graph of the time taken versus n has been executed and verified successfully.

EX.NO:5 IMPLEMENTATION OF NAÏVE PATTERN MATCHING ALGORITHM

11/03/25

AIM:

Given a text txt[0...n-1] and a pattern pat[0...m-1], write a function search (char pat[],char txt[]) that prints all the occurrences of pat[i] in txt[]. You may assume that n>m.

PSEUDOCODE:

PROGRAM:

```
from google.colab import drive

drive.mount('/content/drive')

import time

import matplotlib.pyplot as plt

def naive_pattern_search(pat, txt,nbasicop):

  n = len(txt)

  m = len(pat)

  nbasicop = 0

  occur = []

  for i in range(n - m + 1):

    nbasicop+=1

    j = 0

    while j < m and txt[i + j] == pat[j]:

      nbasicop += 1

      j += 1

    if j == m:

      nbasicop+=1

      occur.append(i)

  return nbasicop

def measure_time(func, *args):

  basicop=0

  start = time.time()

  result = func(*args,basicop)
```

```
    end = time.time()
    return result, end - start
lengths = [1,3,5,8,10,13,15,18,20,23,25,28,30,33,35]
file_path = "/content/drive/MyDrive/input.txt"
with open(file_path, "r") as f:
    lines = f.readlines()
    full_txt = lines[0].strip()
    pat = lines[1].strip()
times = []
ops = []
valid_lengths = []
pat="bbb"
for length in lengths:
    if length >= len(full_txt):
        continue
    txt = full_txt[:length]
    nbasicop,execution = measure_time(naive_pattern_search, pat, txt)
    valid_lengths.append(length)
    ops.append(nbasicop)
    times.append(execution)
print("Execution Times:", times)
print("Basic Operations:", ops)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
```

```
plt.plot(valid_lengths,times, marker='o', color='blue')

plt.title("Execution Time vs Text Length")

plt.xlabel("Text Length (n)")

plt.ylabel("Execution Time (seconds)")

plt.subplot(1, 2, 2)

plt.plot(valid_lengths,ops, color='green',label="Basicop")

n=[i for i in lengths]

plt.plot(n,n,label="O(n) Plot")

plt.title("Performance of Naive Pattern in terms of nnumber of Basic Operations")

plt.xlabel("Text Length (n)")

plt.ylabel("Number of Basic Operations")

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

BEST CASE:

Execution Times: [2.86102294921875e-06, 2.384185791015625e-06, 2.1457672119140625e-06, 5.9604644775390625e-06, 2.384185791015625e-06, 2.1457672119140625e-06, 2.384185791015625e-06, 2.86102294921875e-06, 2.86102294921875e-06, 2.86102294921875e-06, 3.5762786865234375e-06, 3.814697265625e-06, 3.814697265625e-06, 4.291534423828125e-06, 4.76837158203125e-06]

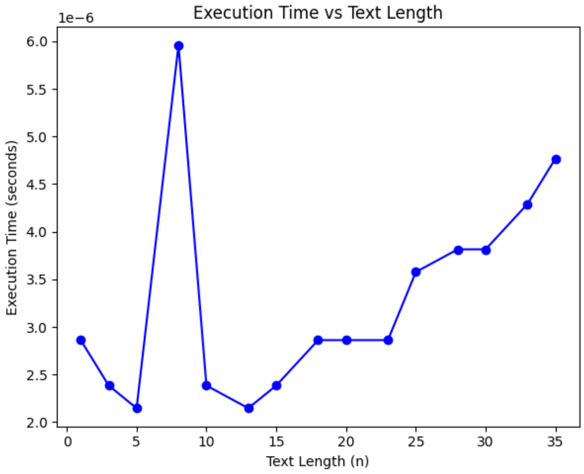
Basic Operations: [0, 1, 3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33]

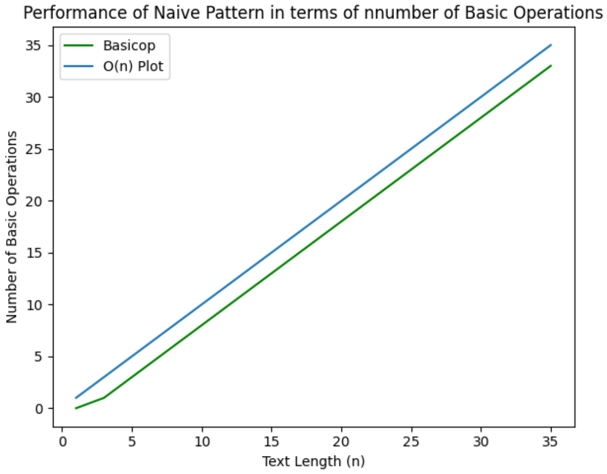
WORST CASE:

Execution Times: [4.76837158203125e-06, 2.6226043701171875e-06, 5.0067901611328125e-06, 3.5762786865234375e-06, 2.86102294921875e-06, 2.86102294921875e-06, 3.5762786865234375e-06, 4.5299530029296875e-06, 4.0531158447265625e-06, 3.814697265625e-06, 4.76837158203125e-06, 4.5299530029296875e-06, 5.0067901611328125e-06, 5.4836273193359375e-06, 5.9604644775390625e-06]

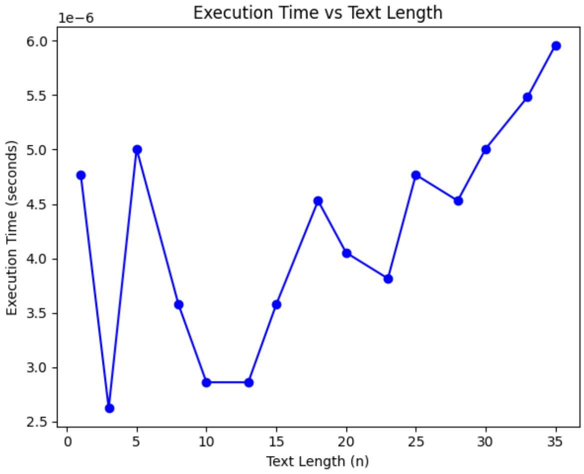
Basic Operations: [1, 3, 5, 8, 10, 13, 15, 18, 20, 23, 25, 28, 30, 33, 35]

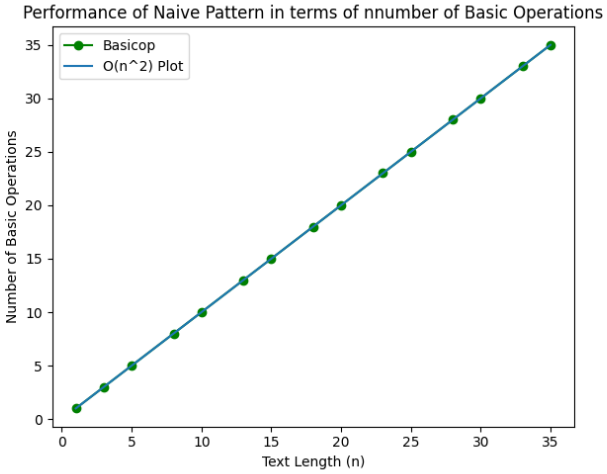
BEST CASE:





WORST CASE:





RESULT:

Thus the program to print all the occurrences of pat[] in txt[] (Naïve Pattern Matching Algorithm) has been executed and verified successfully.

EX.NO:6 IMPLEMENTATION OF RABIN KARP PATTERN

11/03/25 MATCHING ALGORITHM

AIM:

Given a text txt[0…n-1] and a pattern pat[0…m-1], write a function search (char pat[],char txt[]) that prints all the occurrences of pat[i] in txt[]. You may assume that n>m.

PSEUDOCODE:

PROGRAM:

```
def rabin_karp(pat, txt, nbasicop, q=101):
  d = 256
  m = len(pat)
  n = len(txt)
  p = t = 0
  h = 1
  for i in range(m - 1):
    h = (h * d) % q
  for i in range(m):
    p = (d * p + ord(pat[i])) % q
    t = (d * t + ord(txt[i])) % q
  for i in range(n - m + 1):
    nbasicop += 1
    if p == t:
      if txt[i:i + m] == pat:
        nbasicop += 1
        pass
    if i < n-m:
     t = (d * (t - ord(txt[i]) * h) + ord(txt[i + m])) % q
     nbasicop += 1
     if t < 0:
      t += q
      nbasicop += 1
  return nbasicop
```

```
def measure_time(func, *args):
  basicop=0
  start = time.time()
  result = func(*args,basicop)
  end = time.time()
  return result, end - start
lengths = [5,8,10,13,15,18,20,23,25,28,30,33,35,36,37]
file_path = "/content/drive/MyDrive/input.txt"
with open(file_path, "r") as f:
  lines = f.readlines()
  full_txt = lines[0].strip()
  pat = lines[1].strip()
times = []
ops = []
valid_lengths = []
pat="aaa"
for length in lengths:
  if length >= len(full_txt):
    continue
  txt = full_txt[:length]
  nbasicop,execution = measure_time(rabin_karp, pat, txt)
  valid_lengths.append(length)
  ops.append(nbasicop)
  times.append(execution)
print("Execution Times:", times)
```

```
print("Basic Operations:", ops)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(valid_lengths,times, marker='o', color='blue')

plt.title("Execution Time vs Text Length")

plt.xlabel("Text Length (n)")

plt.ylabel("Execution Time (seconds)")

plt.subplot(1, 2, 2)

plt.plot(valid_lengths,ops, color='green',label="Basicop",marker="o")

n=[i for i in lengths]

j=len(pat)

m=[i*j for i in lengths]

print('NM value:',m)

plt.plot(n,m,label="O(nm) Plot")

plt.title("Performance of Rabin Karp Pattern in terms of nnumber of Basic Operations")

plt.xlabel("Text Length (n)")

plt.ylabel("Number of Basic Operations")

#plt.xscale("log")

#plt.yscale("log")

plt.legend()

plt.tight_layout()

plt.show()
```

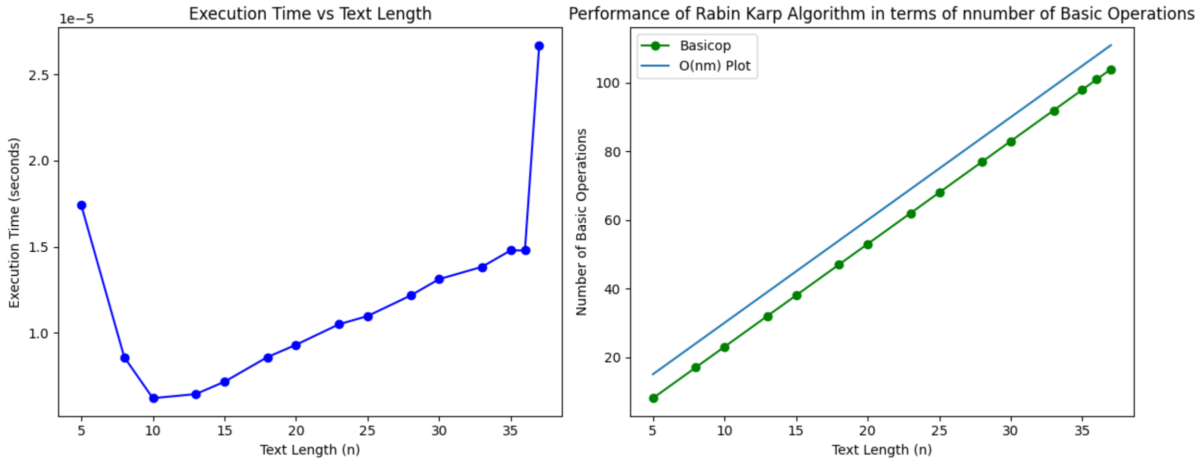
OUTPUT:

WORST CASE:

Execution Times: [1.3589859008789062e-05, 8.344650268554688e-06, 5.4836273193359375e-06, 6.198883056640625e-06, 7.152557373046875e-06, 8.58306884765625e-06, 9.298324584960938e-06, 9.5367431640625e-06, 9.775161743164062e-06, 1.049041748046875e-05, 1.0728836059570312e-05, 1.0967254638671875e-05, 1.2159347534179688e-05, 1.239776611328125e-05, 1.3113021850585938e-05]

Basic Operations: [3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 34, 35]

N value: [15, 24, 30, 39, 45, 54, 60, 69, 75, 84, 90, 99, 105, 108, 111]

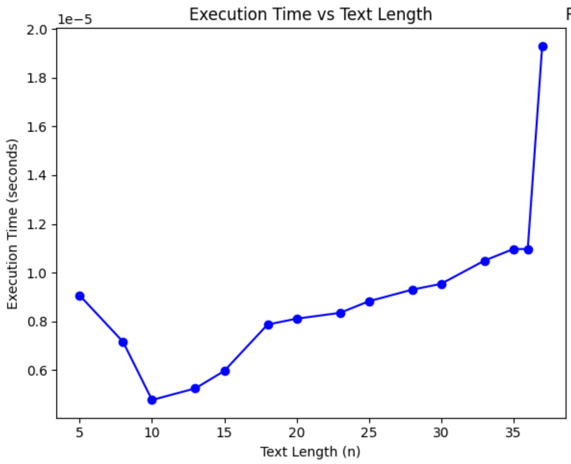


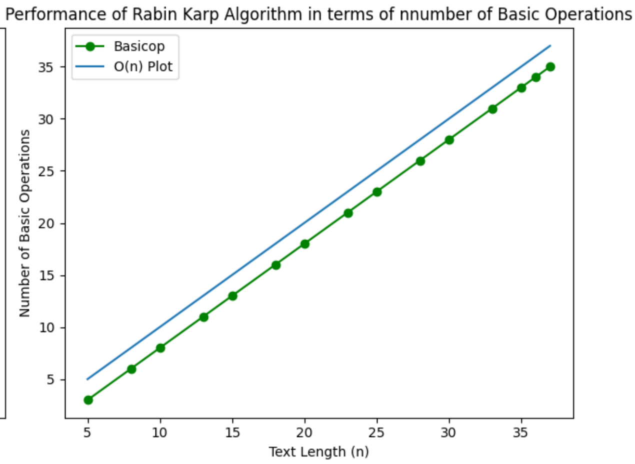
BEST CASE:

Execution Times: [9.059906005859375e-06, 7.152557373046875e-06, 4.76837158203125e-06, 5.245208740234375e-06, 5.9604644775390625e-06, 7.867813110351562e-06, 8.106231689453125e-06, 8.344650268554688e-06, 8.821487426757812e-06, 9.298324584960938e-06, 9.5367431640625e-06, 1.049041748046875e-05, 1.0967254638671875e-05, 1.0967254638671875e-05, 1.9311904907226562e-05]

Basic Operations: [3, 6, 8, 11, 13, 16, 18, 21, 23, 26, 28, 31, 33, 34, 35]

N value: [5, 8, 10, 13, 15, 18, 20, 23, 25, 28, 30, 33, 35, 36, 37]





RESULT:

Thus the program to print all the occurrences of pat[] in txt[] (Rabin Karp Pattern Matching Algorithm) has been executed and verified successfully.

EX.NO:7 ***IMPLEMENTATION OF KNUTH MORRIS PRATT PATTERN***

18/05/25 ***MATCHING ALGORITHM***

AIM:

Given a text txt[0…n-1] and a pattern pat[0…m-1], write a function search (char pat[],char txt[]) that prints all the occurrences of pat[i] in txt[]. You may assume that n>m.

PSEUDOCODE:

PROGRAM:

```
from google.colab import drive

drive.mount('/content/drive')

def compute_lps_array(pat, nbasicop):
  lps = [0] * len(pat)
  length = 0
  i = 1
  while i < len(pat):
    nbasicop += 1
    if pat[i] == pat[length]:
      length += 1
      lps[i] = length
      i += 1
    else:
      if length != 0:
        length = lps[length - 1]
      else:
        lps[i] = 0
        i += 1
  return lps, nbasicop

import time
import matplotlib.pyplot as plt

def kmp_search(pat, txt,nbasicop):
  m = len(pat)
  n = len(txt)
```

```
    lps, nbasicop = compute_lps_array(pat, nbasicop)
    i = 0
    j = 0
    while i < n:
        nbasicop += 1
        if pat[j] == txt[i]:
            i += 1
            j += 1
        if j == m:
            j = lps[j - 1]
        elif i < n and pat[j] != txt[i]:
          if j != 0:
            j = lps[j - 1]
          else:
            i += 1
    return nbasicop
def measure_time(func, *args):
    basicop=0
    start = time.time()
    result = func(*args,basicop)
    end = time.time()
    return result, end - start
lengths = [10,15,20,25,30,35,40,45,50,55,60,65,70]
file_path ="/content/drive/MyDrive/input2.txt"
with open(file_path, "r") as f:
```

```
    lines = f.readlines()
    full_txt = lines[0].strip()
times = []
ops = []
valid_lengths = []
pat="aa"# worst case
pat=”bbb” # best case
for length in lengths:
    if length >= len(full_txt):
        continue
    txt = full_txt[:length]
    nbasicop,execution = measure_time(kmp_search, pat, txt)
    valid_lengths.append(length)
    ops.append(nbasicop)
    times.append(execution)
print("Execution Times:", times)
print("Basic Operations:", ops)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(valid_lengths,times, marker='o', color='blue')
plt.title("Execution Time vs Text Length")
plt.xlabel("Text Length (n)")
plt.ylabel("Execution Time (seconds)")
plt.subplot(1, 2, 2)
plt.plot(valid_lengths,ops, color='green',label="Basicop")
```

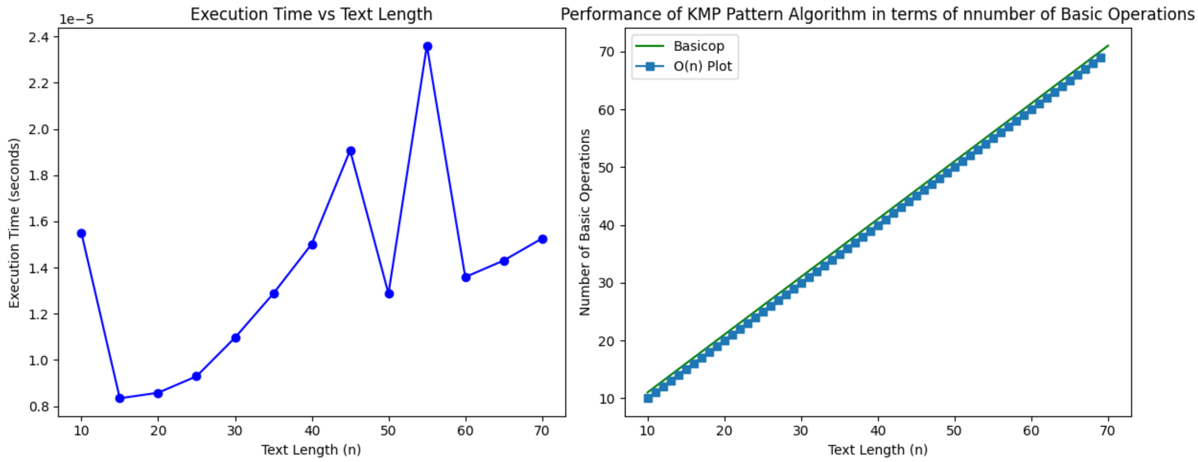
```
n=[i for i in range(10,70)]

j=len(pat)

m=[i for i in range(10,70)]

print('N value:',n)

plt.plot(n,m,label="O(n) Plot",marker="s")

plt.title("Performance of KMP Pattern Algorithm in terms of nnumber of Basic Operations")

plt.xlabel("Text Length (n)")

plt.ylabel("Number of Basic Operations")

#plt.xscale("log")

#plt.yscale("log")

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

WORST CASE:

Execution Times: [1.5497207641601562e-05, 8.344650268554688e-06, 8.58306884765625e-06, 9.298324584960938e-06, 1.0967254638671875e-05, 1.2874603271484375e-05, 1.5020370483398438e-05, 1.9073486328125e-05, 1.2874603271484375e-05, 2.3603439331054688e-05, 1.3589859008789062e-05, 1.430511474609375e-05, 1.52587890625e-05]

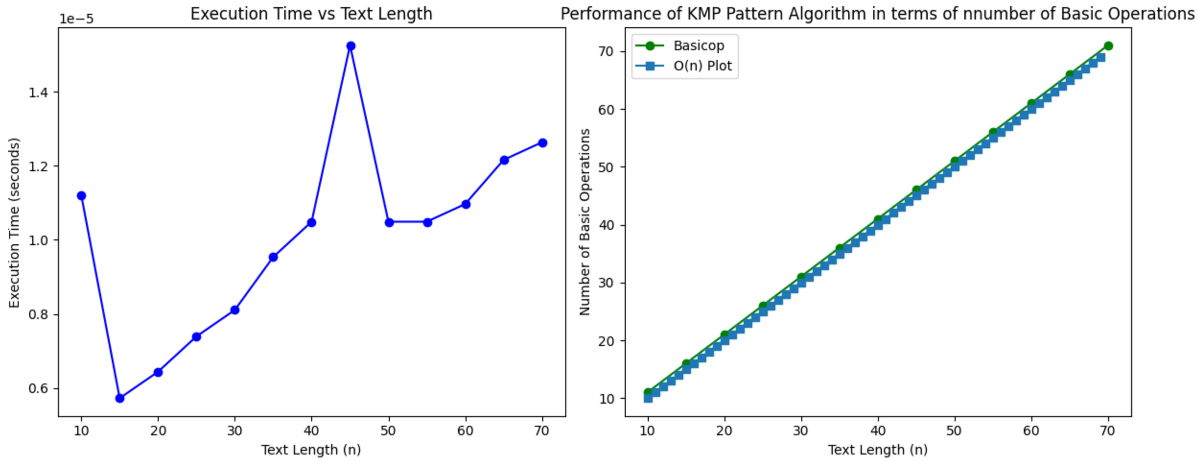
Basic Operations: [11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71]



BEST CASE:

Execution Times: [1.1205673217773438e-05, 5.7220458984375e-06, 6.4373016357421875e-06, 7.3909759521484375e-06, 8.106231689453125e-06, 9.5367431640625e-06, 1.049041748046875e-05, 1.52587890625e-05, 1.049041748046875e-05, 1.049041748046875e-05, 1.0967254638671875e-05, 1.2159347534179688e-05, 1.2636184692382812e-05]

Basic Operations: [11, 16, 21, 26, 31, 36, 41, 46, 51, 56, 61, 66, 71]



RESULT:

Thus the program to print all the occurrences of pat[] in txt[] (Knuth Morris Prattt Pattern Matching Algorithm) has been executed and verified successfully.

EX.NO:8 ***IMPLEMENTATION OF INSERTION SORT***

25/03/25

AIM:

Sort a given set of elements using the Insertion Sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of time taken vs n.

PSEUDOCODE:

PROGRAM:

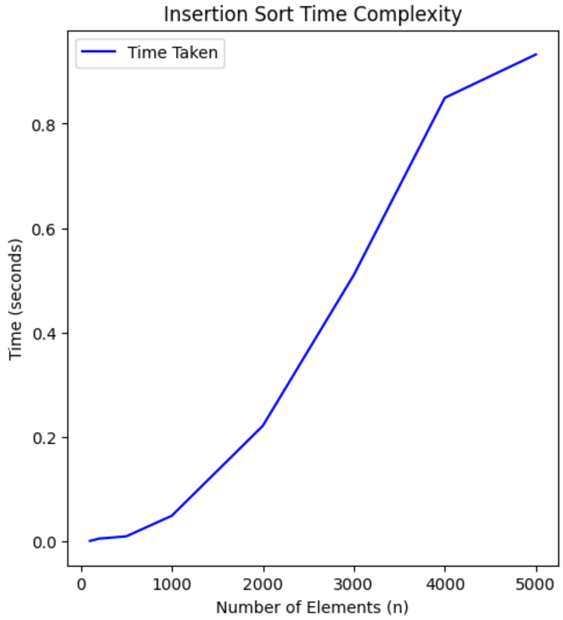
```
import time
import random
import matplotlib.pyplot as plt
def insertion_sort(arr):
    nbasicop = 0
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
            nbasicop += 1
        arr[j + 1] = key
        nbasicop += 1
    return nbasicop
def measure_time_and_ops(arr):
    start_time = time.time()
    nbasicop = insertion_sort(arr)
    end_time = time.time()
    return end_time - start_time, nbasicop
sizes = [100, 200, 500, 1000, 2000, 3000, 4000, 5000]
times = []
basic_ops = []
for n in sizes:
```

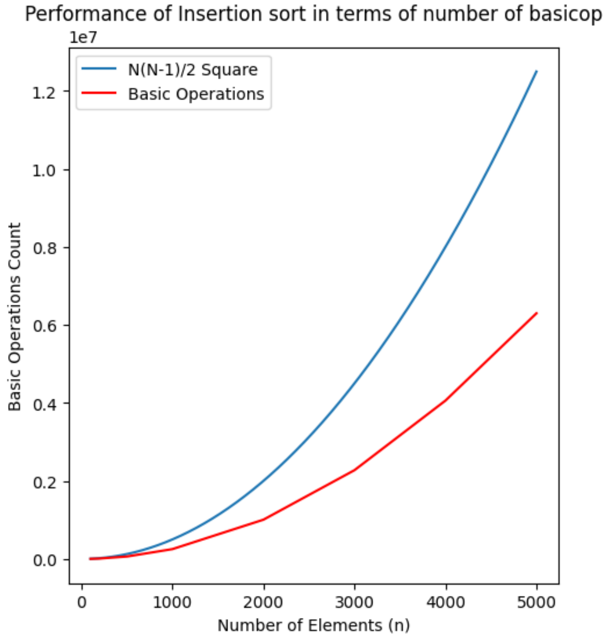
```
        arr = [random.randint(1, 10000) for _ in range(n)]  # Generate random list
        time_taken, nbasicop = measure_time_and_ops(arr)  # Measure time & ops
        times.append(time_taken)
        basic_ops.append(nbasicop)
plt.plot(sizes, times, linestyle='-', color='b', label="Time Taken")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time (seconds)")
plt.title("Insertion Sort Time Complexity")
plt.legend()
n=[i for i in range(100,5000)]
m=[i*(i+1)/2 for i in range(100,5000)]
plt.plot(n,m,label="N(N-1)/2 Square")
plt.plot(sizes, basic_ops, linestyle='-', color='r', label="Basic Operations")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Basic Operations Count")
plt.title("Performance of Insertion Sort in terms of number of basic operation ")
plt.legend()
plt.show()
```

<u>OUTPUT:</u>

Basicop: [2591, 10134, 62083, 251822, 1006531, 2275534, 4060740, 6299324]

Time: [0.00040531158447265625, 0.004895687103271484, 0.009298563003540039, 0.04873299598693848, 0.22108817100524902, 0.5102050304412842, 0.8496370315551758, 0.9326510429382324]





RESULT:

Thus the program to sort the given set of elements using Insertion Sort method and determine the time required to sort the elements for different values of n, the number of elements in the list to be sorted and to plot a graph of the time taken vs n has been executed and verified successfully.

EX.NO:9 IMPLEMENTATION OF HEAP SORT

25/03/25

AIM:

Sort a given set of elements using the Insertion Sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of time taken vs n.

PSEUDOCODE:

PROGRAM:

```
import time

import random

import matplotlib.pyplot as plt

def heapify(arr, n, i, nbasicop):

    largest = i

    left = 2 * i + 1

    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:

      largest = left

      nbasicop += 1

    if right < n and arr[right] > arr[largest]:

        largest = right

        nbasicop += 1

    if largest != i:

        arr[i], arr[largest] = arr[largest], arr[i]

        nbasicop += 1

        nbasicop = heapify(arr, n, largest, nbasicop)

    return nbasicop

def heap_sort(arr):

     n = len(arr)

     nbasicop = 0

     for i in range(n // 2 - 1, -1, -1):

         nbasicop = heapify(arr, n, i, nbasicop)

     for i in range(n - 1, 0, -1):
```

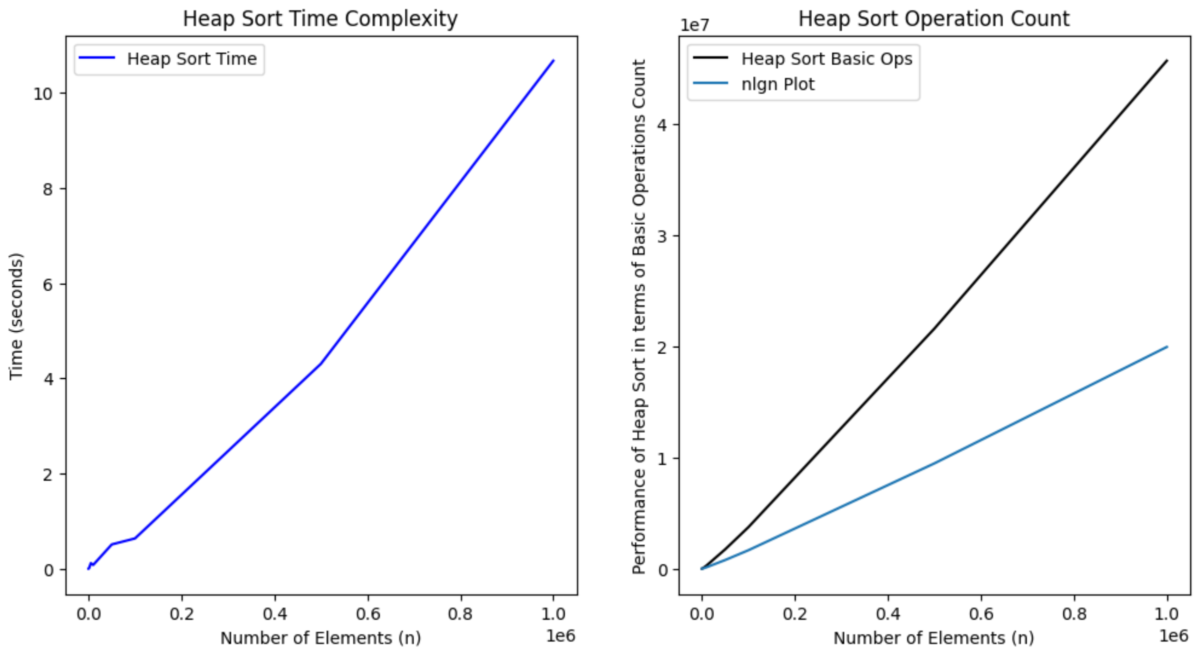
```
        arr[i], arr[0] = arr[0], arr[i]
        nbasicop += 1
        nbasicop = heapify(arr, i, 0, nbasicop)
    return nbasicop
def measure_time_and_ops(sort_function, arr):
    start_time = time.time()
    nbasicop = sort_function(arr)
    end_time = time.time()
    return end_time - start_time, nbasicop
sizes = [10,50,100,500,1000,5000,10000,50000,100000,500000,1000000]
times_heap = []
ops_heap = []
for n in sizes:
  arr = [random.randint(100, 10000000) for _ in range(n)]
  time_taken, nbasicop = measure_time_and_ops(heap_sort, arr)
  times_heap.append(time_taken)
  ops_heap.append(nbasicop)
print("Basicop:",ops_heap)
print("Times:",times_heap)
plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.plot(sizes, times_heap,linestyle='-', color='b', label="Heap Sort Time")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time (seconds)")
plt.title("Heap Sort Time Complexity")
```

```
plt.legend()

plt.subplot(1,2,2)

plt.plot(sizes, ops_heap,"k",label="Heap Sort Basic Ops")

import numpy as np

logvalue=[n * np.log2(n) for n in sizes]

plt.plot(sizes,logvalue,label="nlgn Plot")

plt.xlabel("Number of Elements (n)")

plt.ylabel("Performance of Heap Sort in terms of Basic Operations Count")

plt.title("Heap Sort Operation Count")

plt.legend()

plt.show()
```

OUTPUT:

Basicop: [50, 505, 1254, 9160, 20739, 133244, 290836, 1742990, 3737436, 21582966, 45665358]

Times: [2.956390380859375e-05, 0.00012636184692382812, 0.00030231475830078125, 0.002117156982421875, 0.005112886428833008, 0.1169281005859375, 0.07838916778564453, 0.5093860626220703, 0.6356868743896484, 4.305522680282593, 10.675910234451294]



RESULT:

Thus the program to sort the given set of elements using Insertion Sort method and determine the time required to sort the elements for different values of n, the number of elements in the list to be sorted and to plot a graph of the time taken vs n has been executed and verified successfully.

EX.NO:10 IMPLENTATION OF BREADTH FIRST SEARCH(BFS)
01/04/25

AIM:

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

PSEUDOCODE:

PROGRAM:

```
import time as tm
import matplotlib.pyplot as plt
import random
import numpy as np
from collections import deque
def bfs(graph, start):
    visited = {u: False for u in graph}
    parent = {u: None for u in graph}
    distance = {u: float('inf') for u in graph}
    nbasicop = 0
    queue = deque()
    visited[start] = True
    distance[start] = 0
    queue.append(start)
    while queue:
        u = queue.popleft()
        nbasicop += 1
        for v in graph[u]:
            nbasicop += 1
            if not visited[v]:
                visited[v] = True
                parent[v] = u
                distance[v] = distance[u] + 1
                queue.append(v)
                nbasicop += 1
    return distance, parent, nbasicopdef measure_bfs_time(graph):
    start_node = next(iter(graph))
    start = tm.time()
    _, _, ops = bfs(graph, start_node)
    end = tm.time()
    elapsed_time = end - start
    return ops, elapsed_time
def generate_sparse_graph(n, edge_probability=0.2):
```

```
    graph = {str(i): [] for i in range(n)}
    for i in range(n):
        for j in range(i + 1, n):
            if random.random() < edge_probability:
                graph[str(i)].append(str(j))
                graph[str(j)].append(str(i))
    return graph
def generate_dense_graph(n):
    graph = {str(i): [] for i in range(n)}
    for i in range(n):
        for j in range(n):
            if i != j and random.random() < 0.9:  # 90% edge probability for dense
                graph[str(i)].append(str(j))
    return graph
ns = list(range(2,500, 10))
sparse_times = []
sparse_ops = []
dense_times = []
dense_ops = []
for n in ns:
    sparse_graph = generate_sparse_graph(n)
    sparse_ops_count, sparse_elapsed_time = measure_bfs_time(sparse_graph)
    sparse_times.append(sparse_elapsed_time)
    sparse_ops.append(sparse_ops_count)
    dense_graph = generate_dense_graph(n)
    dense_ops_count, dense_elapsed_time = measure_bfs_time(dense_graph)
    dense_times.append(dense_elapsed_time)
    dense_ops.append(dense_ops_count)
time={dense_elapsed_time:.6f}s, ops={dense_ops_count}")
print("Basicop(sparse):",sparse_ops)
print("Basicop(dense):",dense_ops)
print("Times(sparse)",sparse_times)
print("Times(dense)",dense_times)
plt.figure(figsize=(12, 5))
```

```
plt.subplot(1, 2, 1)
plt.plot(ns, sparse_times, color='blue', label="Sparse Graph")
plt.plot(ns, dense_times, color='green', label="Dense Graph")
plt.title("BFS: n vs Time")
plt.xlabel("Number of Vertices (n)")
plt.ylabel("Time (seconds)")
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(ns, sparse_ops, color='blue', label="Sparse Graph")
n=[i for i in range(2,500)]
m=[(i*i-i)/2 for i in range(2,500)]
plt.plot(n,m,label='V(v-1)/2 Plot')
#plt.plot(ns, dense_ops, marker='s', color='green', label="Dense Graph")
plt.title("BFS: n vs Basic Operations")
plt.xlabel("Number of Vertices (n)")
plt.ylabel("Basic Operations (nbasicop)")
plt.legend()
plt.tight_layout()
plt.show()
```

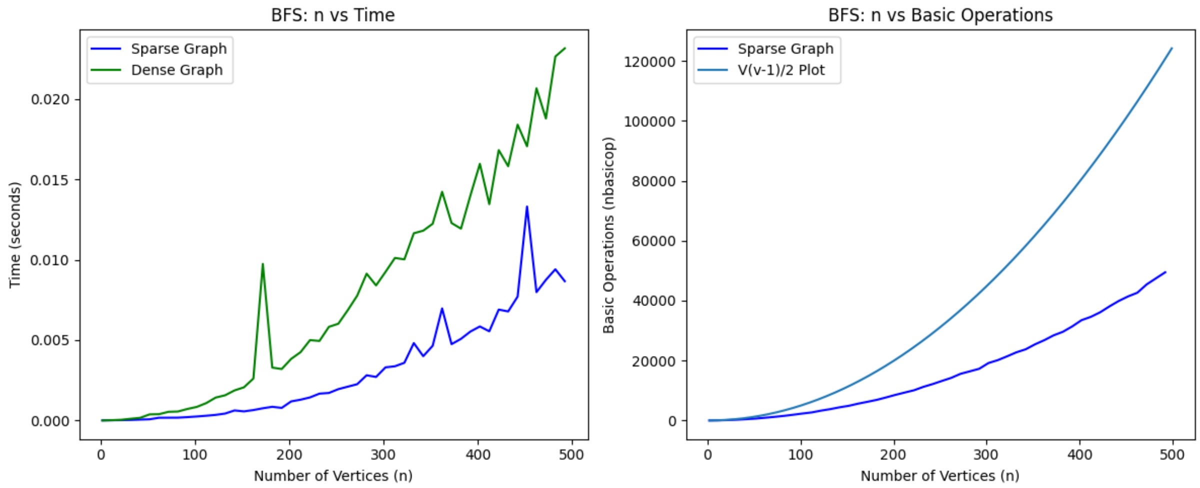
OUTPUT:

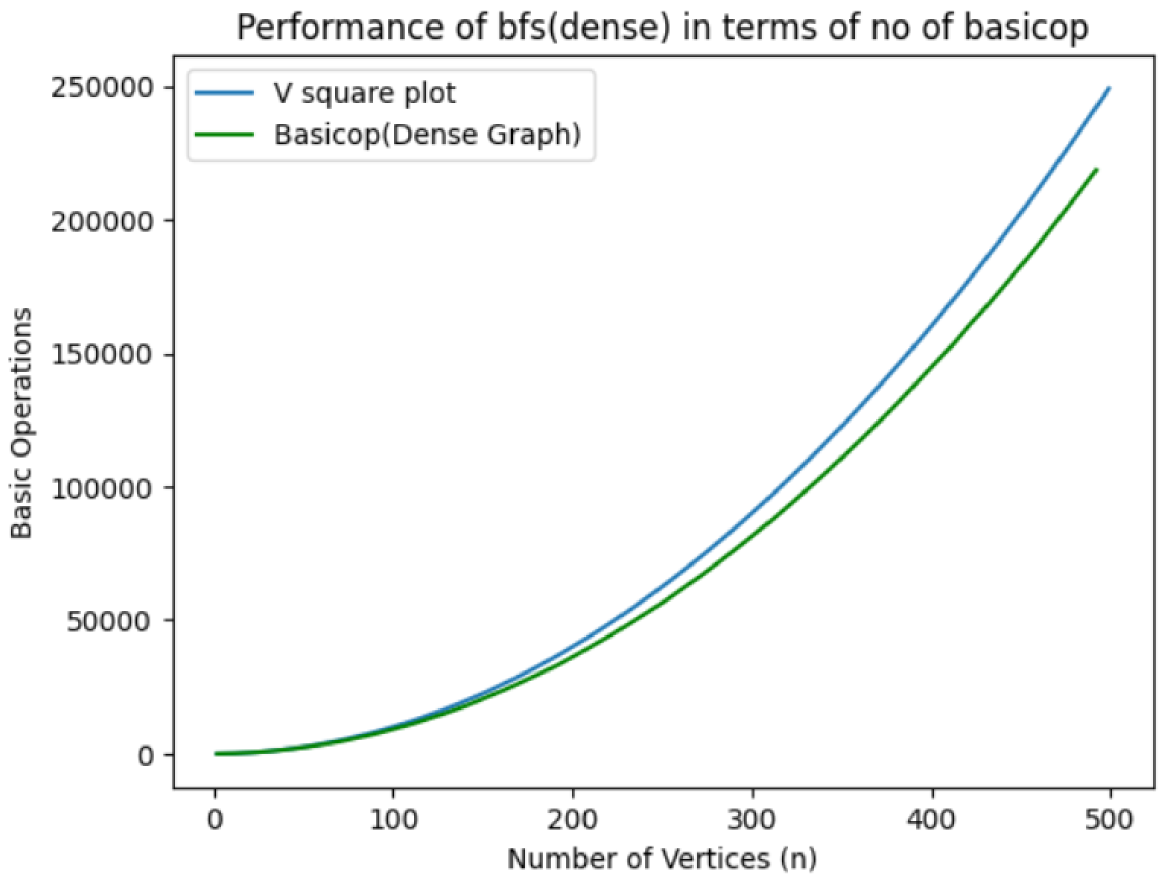
Basicop(sparse): [1, 49, 129, 259, 437, 633, 911, 1203, 1527, 1907, 2313, 2683, 3279, 3785, 4395, 4899, 5615, 6235, 6881, 7691, 8543, 9323, 10085, 11219, 12151, 13199, 14229, 15575, 16407, 17251, 19173, 20137, 21415, 22751, 23763, 25419, 26807, 28419, 29609, 31397, 33485, 34589, 36111, 38081, 39893, 41411, 42647, 45415, 47457, 49479]

Basicop(dense): [5, 141, 462, 972, 1652, 2495, 3550, 4742, 6148, 7702, 9491, 11396, 13566, 15809, 18236, 21098, 23816, 26919, 30012, 33268, 36895, 40734, 44570, 48764, 52961, 57322, 62094, 66789, 71866, 77138, 82458, 87866, 93668, 99582, 105613, 111907, 118318, 124948, 131738, 138888, 146035, 153087, 160988, 168287, 176201, 184218, 192408, 201064, 209684, 218525]

Times(sparse) [1.2159347534179688e-05, 1.9550323486328125e-05, 2.4318695068359375e-05, 3.600120544433594e-05, 5.936622619628906e-05, 7.82012939453125e-05, 0.00017404556274414062, 0.00017690658569335938, 0.000179290771484375, 0.00021219253540039062,.......]

Times(dense) [4.5299530029296875e-06, 2.1696090698242188e-05, 4.649162292480469e-05, 0.000110626220703125, 0.00016832351684570312, 0.0003819465637207031, 0.0003952980041503906, 0.0005412101745605469, 0.0005605220794677734, ...]





RESULT:

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

EX.NO:11 IMPLEMENTATION OF DEPTH FIRST SEARCH(DFS)

01/04/25

AIM:

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

PSEUDOCODE:

PROGRAM:

```
import time as tm

import matplotlib.pyplot as plt

import random

import numpy as np

def dfs(graph,nbasicop):

  visited = {u: False for u in graph}

  parent = {u: None for u in graph}

  discovery_time = {}

  finish_time = {}

  for u in graph:

    nbasicop+=1

    if not visited[u]:  # Avoid repeat traversal

      nbasicop = dfs_visit(graph, u, visited, parent, discovery_time, finish_time, nbasicop)

  return discovery_time, finish_time, parent, nbasicop

def dfs_visit(graph, u, visited, parent, discovery_time, finish_time, nbasicop):

  discovery_time[u] = nbasicop

  visited[u] = True

  for v in graph[u]:

    nbasicop += 1

    if not visited[v]:

      parent[v] = u

      nbasicop = dfs_visit(graph, v, visited, parent, discovery_time, finish_time, nbasicop)

  finish_time[u] = nbasicop
```

```
    return nbasicop
def measure_time(graph):
    nbasicop=0
    start = tm.time()
    _, _, _, ops = dfs(graph,nbasicop)
    end = tm.time()
    elapsed_time = end - start
    return ops, elapsed_time
def generate_sparse_graph(n, edge_probability=0.2):
    graph = {str(i): [] for i in range(n)}
    for i in range(n):
        for j in range(i + 1, n):
            if random.random() < edge_probability:
                graph[str(i)].append(str(j))
                graph[str(j)].append(str(i))
    return graph
def generate_dense_graph(n):
    matrix = np.random.randint(0, 2, size=(n, n))
    np.fill_diagonal(matrix, 0)
    return matrix
def adjacency_matrix_to_list(matrix):
    n = matrix.shape[0]
    graph = {str(i): [] for i in range(n)}
    for i in range(n):
        for j in range(n):
```

```
            if matrix[i][j] == 1:
                graph[str(i)].append(str(j))
    return graph
ns = list(range(1,100, 10))
sparse_times = []
sparse_ops = []
dense_times = []
dense_ops = []
for n in ns:
  sparse_graph = generate_sparse_graph(n)
  sparse_ops_count, sparse_elapsed_time = measure_time(sparse_graph)
  sparse_times.append(sparse_elapsed_time)
  sparse_ops.append(sparse_ops_count)
  dense_matrix = generate_dense_graph(n)
  dense_graph = adjacency_matrix_to_list(dense_matrix)
  dense_ops_count, dense_elapsed_time = measure_time(dense_graph)
  dense_times.append(dense_elapsed_time)
  dense_ops.append(dense_ops_count)
plt.figure(figsize=(14, 6))
print("Basicop(sparse):",sparse_ops)
print("Basicop(dense):",dense_ops)
print("Times(sparse)",sparse_times)
print("Times(dense)",dense_times)
plt.subplot(1, 2, 1)
plt.plot(ns, sparse_times, color='blue', label="Sparse Graph")
```

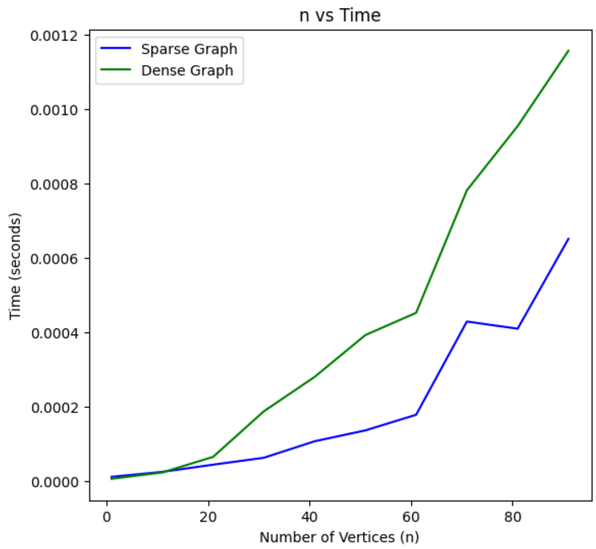
```
plt.plot(ns, dense_times, color='green', label="Dense Graph")

plt.title("n vs Time")

plt.xlabel("Number of Vertices (n)")

plt.ylabel("Time (seconds)")

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(ns, sparse_ops, color='blue', label="Sparse Graph")

plt.title("Performance of DFS in terms of no of basicop")

n=[i for i in range(1,100)]

m=[i*i for i in range(1,100)]

o=[(n*n-n)/2 for n in range(1,100)]

plt.plot(n,o,label='N(N-1)/2 Plot')

plt.xlabel("Number of Vertices (n)")

plt.ylabel("Basic Operations (nbasicop)")

plt.legend()

plt.plot(ns, dense_ops, marker='s', color='green', label="Dense Graph")

n=[i for i in range(1,100)]

m=[i*i for i in range(1,100)]

plt.plot(n,m,label='V Square Plot')

plt.tight_layout()

plt.title("Performance of DFS in terms of no of basicop")

plt.xlabel("Number of Vertices (n)")

plt.ylabel("Basic Operations")

plt.legend()

plt.show()
```

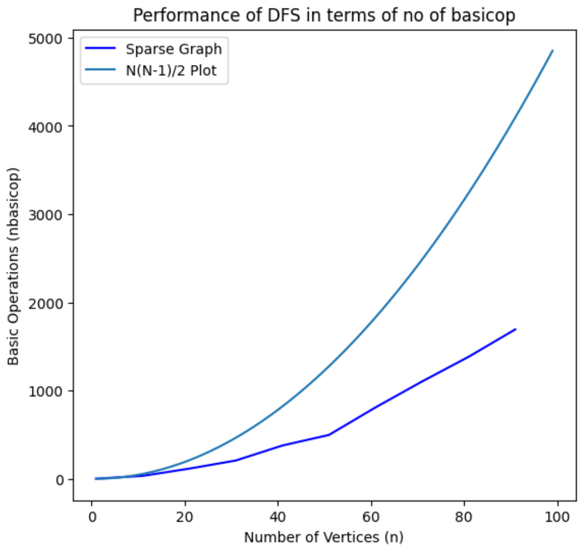
OUTPUT:

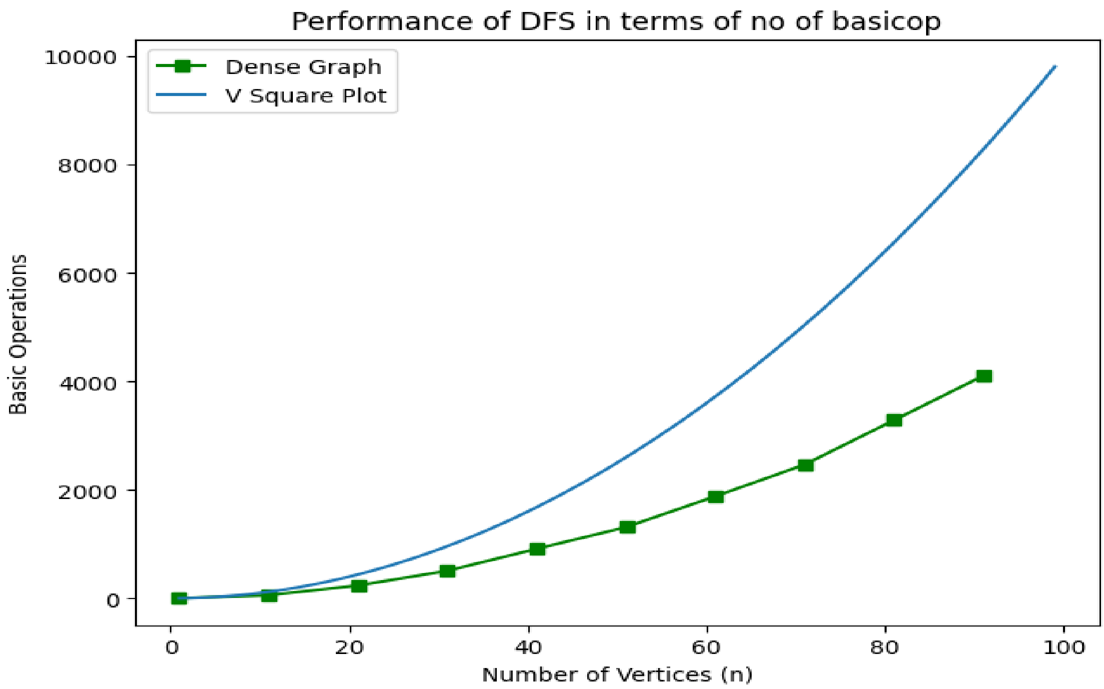
Basicop(sparse): [1, 33, 115, 207, 377, 497, 809, 1103, 1383, 1693]

Basicop(dense): [1, 59, 238, 509, 912, 1314, 1881, 2467, 3285, 4104]Times(sparse) [1.0251998901367188e-05, 2.3603439331054688e-05, 4.291534423828125e-05, 6.151199340820312e-05, 0.00010585784912109375, 0.00013518333435058594, 0.00017714500427246094, 0.0004279613494873047, 0.0004086494445800781, 0.0006501674652099609]

Times(dense) [5.0067901611328125e-06, 2.193450927734375e-05, 6.389617919921875e-05, 0.00018644332885742188, 0.0002789497375488281, 0.0003914833068847656, 0.0004513263702392578, 0.0007808208465576172, 0.0009534358978271484, 0.0011560916900634766]







RESULT:

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

EX.NO:12 IMPLEMENTATION OF DIJKSTRA'S ALGORITHM

08/04/25

AIM:

From the given vertex in a weighted connected graph, develop a program to find the shortest paths to other vertices using the Dijkstra's Algorithm.

PSEUDOCODE:

PROGRAM:

```
import time
import matplotlib.pyplot as plt
import numpy as np
import heapq
def dijkstra_matrix(graph, start):
    n = len(graph)
    visited = [False] * n
    dist = [float('inf')] * n
    dist[start] = 0
    heap = [(0, start)]
    basicop = 0
    while heap:
        d, u = heapq.heappop(heap)
        basicop += 1
        if visited[u]:
            continue
        visited[u] = True
        basicop += 1
        for v in range(n):
            basicop += 1
            if graph[u][v] > 0 and not visited[v]:
                if dist[u] + graph[u][v] < dist[v]:
                    dist[v] = dist[u] + graph[u][v]
                    heapq.heappush(heap, (dist[v], v))
```

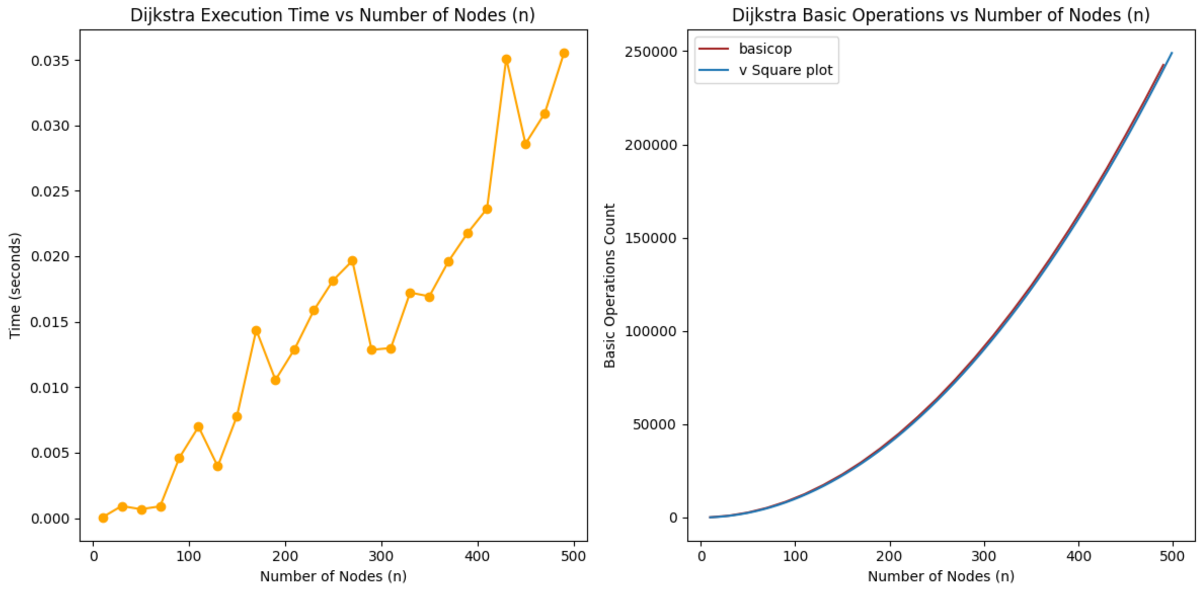
```
                basicop += 1
    return basicop
def measure_time_and_basicop(func, *args):
    start = time.time()
    basicop = func(*args)
    end = time.time()
    return end - start, basicop
def generate_weighted_connected_matrix(n):
    matrix = [[0]*n for _ in range(n)]
    for i in range(n - 1):
        weight = np.random.randint(1, 10)
        matrix[i][i + 1] = matrix[i + 1][i] = weight
    for i in range(n):
        for j in range(i + 2, n):
            if np.random.rand() < 0.05:
                weight = np.random.randint(1, 10)
                matrix[i][j] = matrix[j][i] = weight
    return matrix
sizes = list(range(10,500, 20))
times = []
basicops = []
for size in sizes:
    g = generate_weighted_connected_matrix(size)
    t, basicop = measure_time_and_basicop(dijkstra_matrix, g, 0)
    times.append(t)
```

```
basicops.append(basicop)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(sizes, times, marker='o', color='orange')

plt.title("Dijkstra Execution Time vs Number of Nodes (n)")

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Time (seconds)")

plt.subplot(1, 2, 2)

plt.plot(sizes, basicops,label='basicop', color='brown')

plt.title("Dijkstra Basic Operations vs Number of Nodes (n)")

n=[i for i in range(10,500)]

m=[i*i for i in range(10,500)]

plt.plot(n,m,label='v Square plot')

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Basic Operations Count")

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

Times: [6.341934204101562e-05, 0.0009272098541259766, 0.0006842613220214844, 0.0009016990661621094, 0.004611015319824219, 0.006979942321777344, 0.00395655632019043, 0.007805347442626953, 0.014373540878295898, 0.010559320449829102, 0.012921810150146484, 0.0158841609954834, 0.018167734146118164, 0.019657135009765625, 0.012853622436523438, 0.01299142837524414, 0.017226696014404297, 0.016941547393798828, 0.01961970329284668, 0.021795272827148438, 0.023638010025024414, 0.0351102352142334, 0.028589248657226562, 0.030913114547729492, 0.035523414611816406]

Basic Operation: [131, 997, 2673, 5165, 8439, 12525, 17465, 23135, 29637, 36967, 45027, 53969, 63617, 74197, 85489, 97601, 110565, 124209, 138729, 154073, 170145, 187011, 204857, 223281, 242587]



RESULT:

Thus the program to find the shortest paths to other vertices from a given vertex in a weighted connected graph using Dijkstra's Agorithm has been executed and verified successfully.

EX.NO:13 IMPLEMENTATION OF PRIM'S ALGORITHM

08/04/25

AIM:

To find the minimum cost spanning tree of a given undirected graph using Prim's Algorithm.

PSEUDOCODE:

PROGRAM:

```
import time

import matplotlib.pyplot as plt

import numpy as np

import heapq

def prim_matrix(graph,basicop):

  n = len(graph)

  visited = [False] * n

  min_heap = [(0, 0)]  # (cost, vertex)

  total_cost = 0

  while min_heap:

    cost, u = heapq.heappop(min_heap)

    basicop += 1  # pop from heap

    if visited[u]:

      continue

    visited[u] = True

    total_cost += cost

    for v in range(n):

      basicop += 1  # check neighbor

      if graph[u][v] > 0 and not visited[v]:

        heapq.heappush(min_heap, (graph[u][v], v))

  return basicop

def measure_time_and_basicop(func, *args):

  basicop=0

  start = time.time()
```

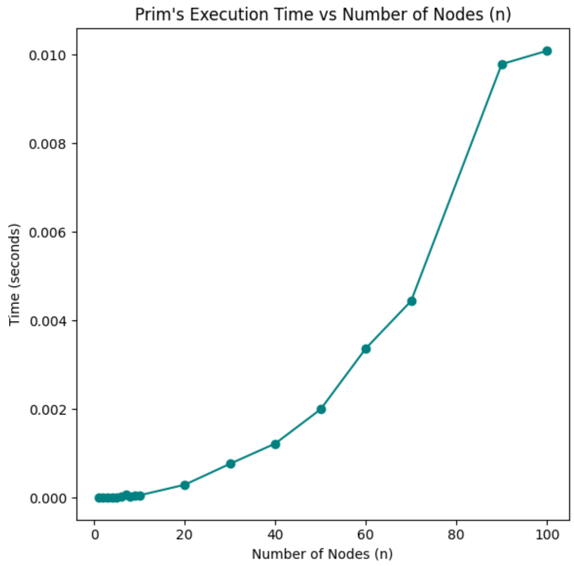
```
    basicop = func(*args,basicop)
    end = time.time()
    return end - start, basicop
def generate_weighted_connected_matrix(n):
    matrix = [[0]*n for _ in range(n)]
    for i in range(n - 1):
        weight = np.random.randint(1, 10)
        matrix[i][i + 1] = matrix[i + 1][i] = weight
    for i in range(n):
        for j in range(i + 2, n):
            if np.random.rand() < 1:
                weight = np.random.randint(1, 10)
                matrix[i][j] = matrix[j][i] = weight
    return matrix
sizes = [1,2,3,4,5,6,7,8,9,10,20,30,40,50,60,70,90,100]
times = []
basicops = []
for size in sizes:
    g = generate_weighted_connected_matrix(size)
    t, basicop = measure_time_and_basicop(prim_matrix, g)
    times.append(t)
    basicops.append(basicop)
print("Basicop:",basicops)
print("Times:",times)
plt.figure(figsize=(12, 6))
```

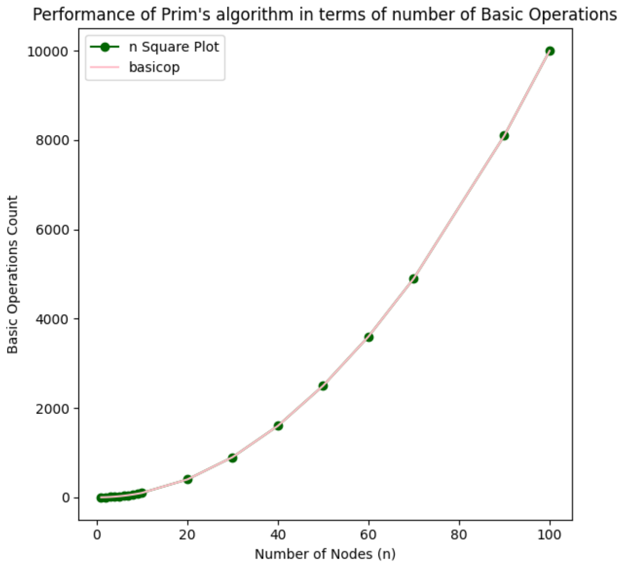
```
plt.subplot(1, 2, 1)

plt.plot(sizes, times, marker='o', color='teal')

plt.title("Prim's Execution Time vs Number of Nodes (n)")

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Time (seconds)")

plt.subplot(1, 2, 2)

n=[i for i in sizes]

m=[i*i for i in sizes]

plt.plot(n,m,label="n Square Plot",color='darkgreen',marker="o")

plt.plot(sizes, basicops, color='pink',label="basicop")

plt.title("Performance of Prim's algorithm in terms of number of Basic Operations")

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Basic Operations Count")

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

Basicop: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 400, 900, 1600, 2500, 3600, 4900, 8100, 10000]

Times: [8.821487426757812e-06, 7.867813110351562e-06, 1.0728836059570312e-05, 8.58306884765625e-06, 1.239776611328125e-05, 1.9311904907226562e-05, 7.605552673339844e-05, 3.314018249511719e-05, 4.410743713378906e-05, 5.221366882324219e-05, 0.0002906322479248047, 0.0007636547088623047, 0.0012204647064208984, 0.001991748809814453, 0.0033676624298095703, 0.004438161849975586, 0.009781837463378906, 0.010082244873046875]





RESULT:

Thus the program to find the minimum cost spanning tree of a given undirected graph using Prim's Algorithm has been executed and verified successfully.

EX.NO:14 IMPLENTATION OF FLOYD WARSHALL'S ALGORITHM

15/04/25

AIM:

To implement Floyd's Algorithm for the All-Shortest-Paths problem.

PSEUDOCODE:

PROGRAM:

```
import time

import matplotlib.pyplot as plt

import numpy as np

def floyd_warshall(graph):

  n = len(graph)

  dist = [[float('inf')] * n for _ in range(n)]

  basicop = 0

  for i in range(n):

    for j in range(n):

      if i == j:

        dist[i][j] = 0

      elif graph[i][j] > 0:

        dist[i][j] = graph[i][j]

  for k in range(n):

    for i in range(n):

      for j in range(n):

        basicop += 1

        if dist[i][k] + dist[k][j] < dist[i][j]:

          dist[i][j] = dist[i][k] + dist[k][j]

          #basicop += 1

  return basicop

def measure_time_and_basicop(func,*args):

  basicop=0

  start = time.time()
```

```
    basicop = func(*args)
    end = time.time()
    return end - start, basicop
def generate_weighted_connected_matrix(n):
    matrix = [[0]*n for _ in range(n)]
    for i in range(n - 1):
        weight = np.random.randint(1, 10)
        matrix[i][i + 1] = matrix[i + 1][i] = weight
    for i in range(n):
        for j in range(i + 2, n):
            if np.random.rand() < 0.1:
                weight = np.random.randint(1, 10)
                matrix[i][j] = matrix[j][i] = weight
    return matrix
sizes = [1,5,10,50,100,500]
times = []
basicops = []
for size in sizes:
    g = generate_weighted_connected_matrix(size)
    t, basicop = measure_time_and_basicop(floyd_warshall, g)
    times.append(t)
    basicops.append(basicop)
print('Basic Operation')
print(basicops)
print('Time')
```

```
print(times)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(sizes, times, color='navy')

plt.title("Floyd-Warshall Execution Time vs Number of Nodes (n)")

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Time (seconds)")

plt.subplot(1, 2, 2)

plt.plot(sizes, basicops, color='crimson',label='Basicop',marker="*")

n=[i for i in range(1,500)]

m=[n*n*n for n in range(1,500)]

plt.plot(n,m,label=' V cube Plot')

plt.xscale('log')

plt.yscale('log')

plt.title("Performance of Flody Warshall Algorithm in terms of no of basicop")

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Basic Operations Count")

plt.legend()

plt.tight_layout()

plt.show()
```

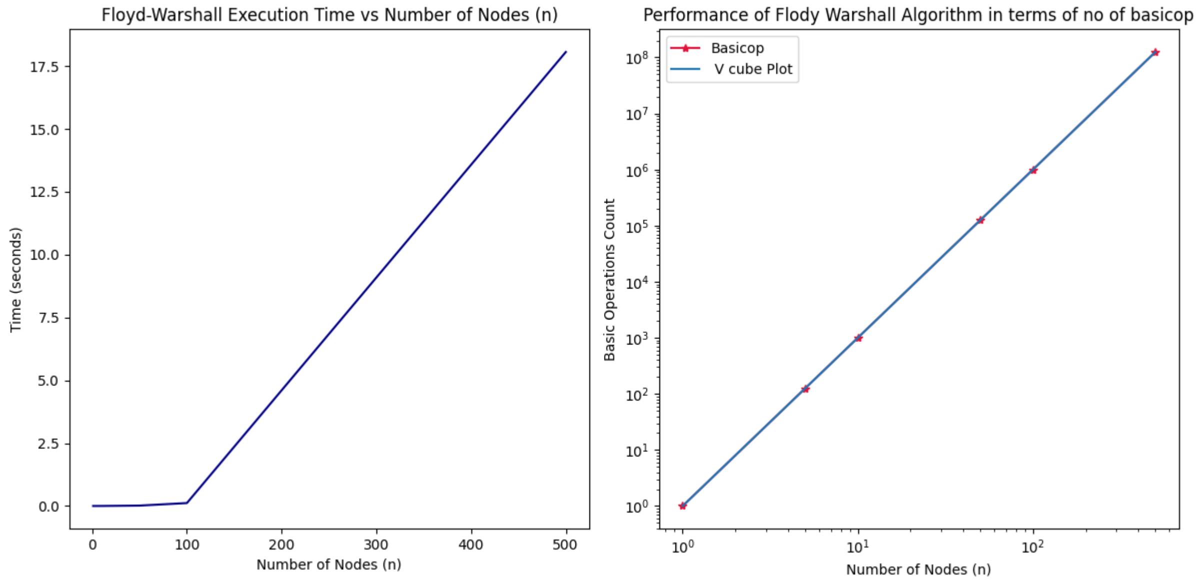
OUTPUT:

Basic Operation

[1, 125, 1000, 125000, 1000000, 125000000]

Time:

[1.2636184692382812e-05, 5.364418029785156e-05, 0.0007991790771484375, 0.015152692794799805, 0.11583709716796875, 18.059677124023438]



RESULT:

Thus the program to implement Floyd's Algorithm for the All-Pairs-Shortest-PathsProblem has been executed and verified successfully.

EX.NO:15 IMPLEMENTATION OF TRANSITIVE CLOSURE OF

15/04/25 CONNECTED GRAPH (WARSHALL'S ALGORITHM)

AIM:

To compute the transitive closure of a given directed graph using the Warshall's Algorithm.

PSEUDOCODE:

PROGRAM:

```
import time
import matplotlib.pyplot as plt
import numpy as np
def transitive_closure_floyd_warshall(W,nbasicop):
    n = len(W)
    D = [[W[i][j] for j in range(n)] for i in range(n)]
    for k in range(n):
        for i in range(n):
            for j in range(n):
                nbasicop+=1
                D[i][j] = D[i][j] or (D[i][k] and D[k][j])
    return nbasicop
def measure_time_and_basicop(func,*args):
    basicop=0
    start = time.time()
    basicop = func(*args,basicop)
    end = time.time()
    return end - start, basicop
def generate_weighted_connected_matrix(n):
    matrix = [[0]*n for _ in range(n)]
    for i in range(n - 1):
        weight = np.random.randint(1, 10)
        matrix[i][i + 1] = matrix[i + 1][i] = weight
    for i in range(n):
```

```
        for j in range(i + 2, n):
            if np.random.rand() < 0.1:
                weight = np.random.randint(1, 10)
                matrix[i][j] = matrix[j][i] = weight
    return matrix
sizes = [1,5,10,50,100,500]
times = []
basicops = []
for size in sizes:
    g = generate_weighted_connected_matrix(size)
    t, basicop = measure_time_and_basicop(transitive_closure_floyd_warshall, g)
    times.append(t)
    basicops.append(basicop)
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sizes, times, color='navy')
plt.title("Floyd-Warshall Execution Time vs Number of Nodes (n)")
plt.xlabel("Number of Nodes (n)")
plt.ylabel("Time (seconds)")
plt.subplot(1, 2, 2)
plt.plot(sizes, basicops, color='crimson',label='Basicop',marker="o")
n=[i for i in range(1,500)]
m=[ n*n*n for n in range(1,500)]
plt.plot(n,m,label=' V cube Plot')
print("Basic Operation:")
```

```
print(basicops)

print("Times:")

print(times)

plt.title("Performance of Transitive Closure in terms of no of basicop")

plt.xlabel("Number of Nodes (n)")

plt.ylabel("Basic Operations Count")

plt.legend()

plt.xscale("log")

plt.yscale("log")

plt.tight_layout()

plt.show()
```

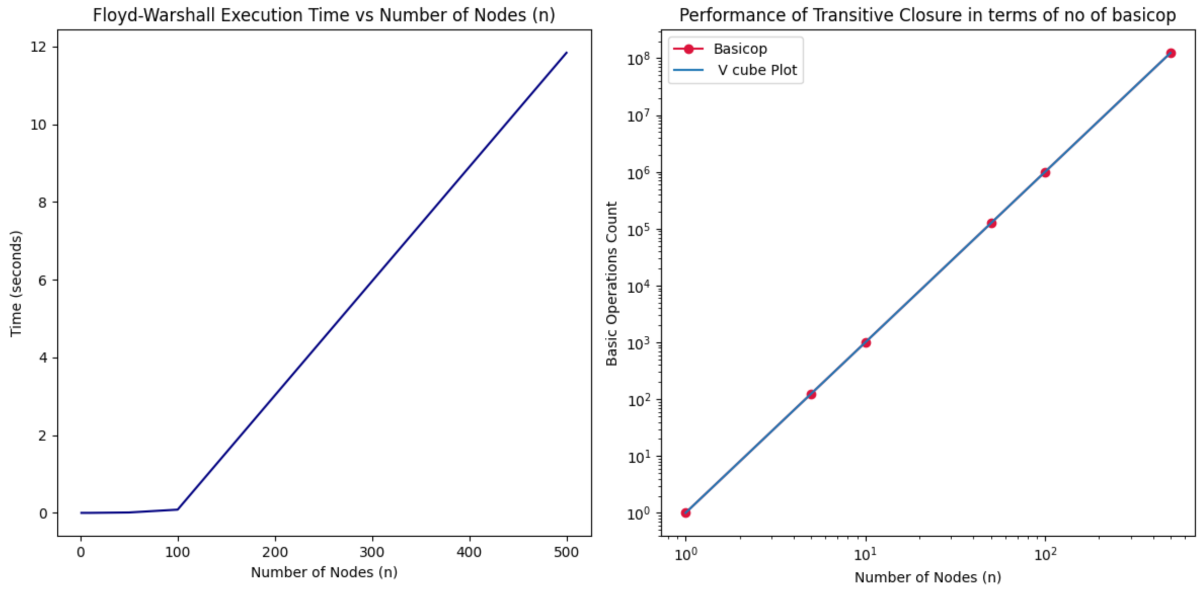
OUTPUT:

Basic Operation:

[1, 125, 1000, 125000, 1000000, 125000000]

Times:

[1.2636184692382812e-05, 2.47955322265625e-05, 0.00011515617370605469, 0.010694742202758789, 0.08321404457092285, 11.83737587928772]



RESULT:

Thus the program to compute the transitive closure of a given directed graph using Warshall's Algorithm has been executed and verified successfully.

EX.NO:16 IMPLEMENTATION OF FINDING MINIMUM AND MAXIMUM

22/05/25 USING DIVIDE AND CONQUER METHOD

AIM:

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

PSEUDOCODE:

PROGRAM:

```
import time

import random

import matplotlib.pyplot as plt

def max_min(arr, i, j,nbasicop):

  if i == j:

    return arr[i], arr[i],nbasicop

  elif i == j - 1:

    nbasicop += 1

    if arr[i] < arr[j]:

      return arr[j], arr[i],nbasicop

    else:

      return arr[i], arr[j],nbasicop

  else:

    mid = (i + j) // 2

    max1, min1,nbasicop = max_min(arr, i, mid,nbasicop)

    max2, min2,nbasicop = max_min(arr, mid + 1, j,nbasicop)

    nbasicop+= 2

    return max(max1, max2), min(min1, min2),nbasicop

def measure_time_and_ops(arr):

  nbasicop = 0

  start_time = time.time()

  max,min,nbasicops=max_min(arr, 0, len(arr) - 1,nbasicop)

  end_time = time.time()

  return end_time - start_time, nbasicops
```

```
def build_best_case(arr):
  if not arr:
    return []
  mid=len(arr)//2
  result = [arr[mid]]
  result.extend(build_best_case(arr[:mid]))
  result.extend(build_best_case(arr[mid+1:]))
  return result
ns = list(range(10,10000, 50))
times = []
ops = []
for n in ns:
    arr = random.sample(range(10, 10000), n)
    ar=build_best_case(arr)
    t, op = measure_time_and_ops(ar)
    times.append(t)
    ops.append(op)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(ns, times, color='blue')
plt.title("Time vs n")
plt.xlabel("n (Input Size)")
plt.ylabel("Time (seconds)")
plt.subplot(1, 2, 2)
plt.plot(ns, ops, label='basicop', color='green',marker="*")
```

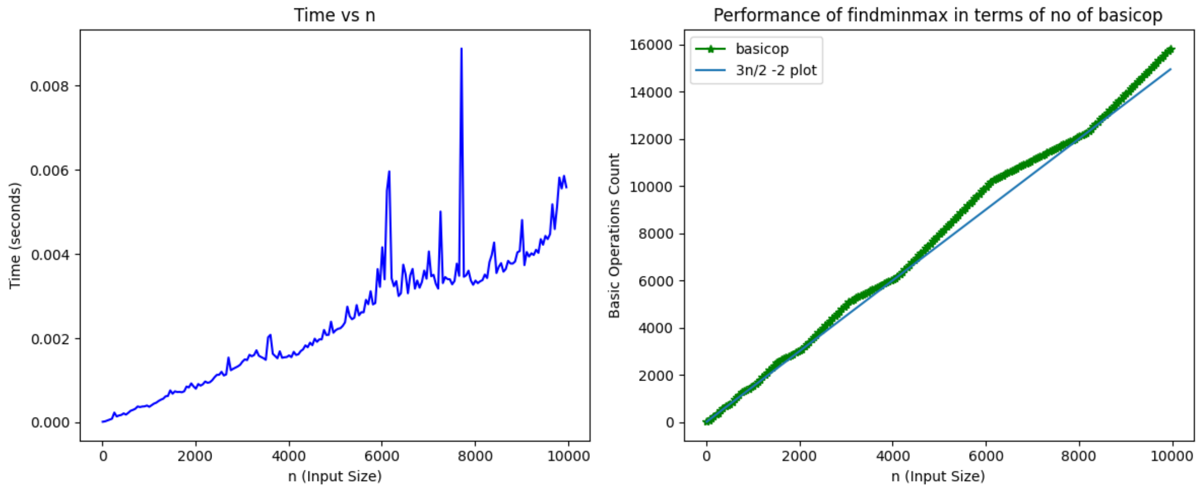
```
plt.title("Performance of findminmax in terms of no of basicop")

plt.xlabel("n (Input Size)")

plt.ylabel("Basic Operations Count")

n=[i for i in ns]

m=[1.5*i for i in ns]

plt.plot(n,m,label='3n/2 -2 plot')

#plt.xscale('log')

#plt.yscale('log')

print('Basic Operation:',ops)

print("Times:",times)

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

BEST CASE:

Basic Operation: [14, 90, 172, 254, 336, 390, 490, 590, 664, 714, 764, 862, 962, 1062, 1162, 1262, 1320, 1370, 1420, 1470, 1520, 1606, 1706, 1806, 1906, 2006, 2106, 2206, 2306, 2406, 2506, 2582, 2632, 2682, 2732, 2782, 2832, 2882, 2932, 2982, 3032, 3094, 3194, 3294, 3394, 3494, 3594, 3694, 3794, 3894, 3994, 4094, 4194, 4294, 4394, 4494, 4594, 4694, 4794, 4894, 4994, 5094.....]

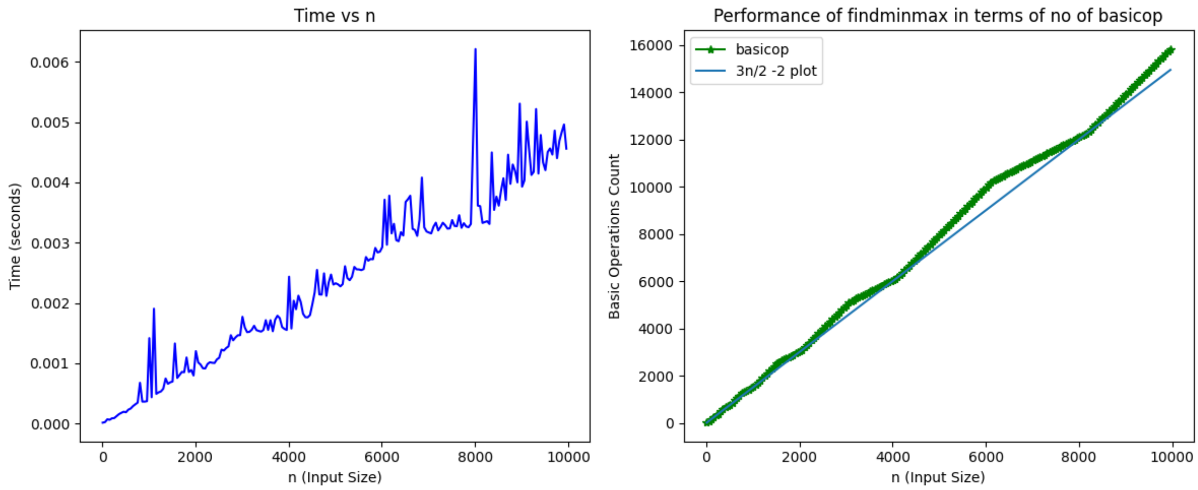
Times: [1.3828277587890625e-05, 2.2649765014648438e-05, 4.3392181396484375e-05, 6.127357482910156e-05, 8.559226989746094e-05, 0.0002315044403076172, 0.0001392364501953125, 0.00016260147094726562, 0.00017571449279785156, 0.0002124309539794922, ...]



WORST CASE:

Basic Operation: [14, 90, 172, 254, 336, 390, 490, 590, 664, 714, 764, 862, 962, 1062, 1162, 1262, 1320, 1370, 1420, 1470, 1520, 1606, 1706, 1806, 1906, 2006, 2106, 2206, 2306, 2406, 2506, 2582, 2632, 2682, 2732, 2782, 2832, 2882, 2932, 2982, 3032, 3094, 3194, 3294, 3394, 3494, 3594, 36......]

Times: [1.4066696166992188e-05, 2.3603439331054688e-05, 7.009506225585938e-05, 6.222724914550781e-05, 8.630752563476562e-05, 9.083747863769531e-05, 0.0001227855682373047, 0.0001556873321533203, 0.00017690658569335938, 0.0001952648162841797, 0.000186920166015625, 0.00022912025451660156, 0.00024819374084472656, ...]



RESULT:

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

EX.NO:17 *IMPLEMENTATION OF MERGE SORT*

26/04/25

AIM:

To implement Merge Sort methods to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

PSEUDOCODE:

PROGRAM:

```
import time
import random
import matplotlib.pyplot as plt
nbasicop = 0
def merge(arr, left, mid, right):
    global nbasicop
    n1 = mid - left + 1
    n2 = right - mid
    L = arr[left:mid + 1]
    R = arr[mid + 1:right + 1]
    i = j = 0
    k = left
    while i < n1 and j < n2:
        nbasicop += 1  # Comparison
        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 merge_sort(arr, left, right):
    if left < right:
      mid = (left + right) // 2
      merge_sort(arr, left, mid)
      merge_sort(arr, mid + 1, right)
      merge(arr, left, mid, right)
def measure_time_and_ops(arr):
    global nbasicop
    nbasicop = 0
    start_time = time.time()
    merge_sort(arr, 0, len(arr) - 1)
    end_time = time.time()
    return end_time - start_time, nbasicop
def build_best_case(arr):
  if not arr:
    return []
  mid=len(arr)//2
  result = [arr[mid]]
  result.extend(build_best_case(arr[:mid]))
  result.extend(build_best_case(arr[mid+1:]))
```

```
  return result
ns =[10,100,1000,10000]
times = []
ops = []
for n in ns:
   ar = random.sample(range(1, 100000), n)
   arr=build_best_case(ar)
   t, op = measure_time_and_ops(arr[:])
   times.append(t)
   ops.append(op)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(ns, times,label='Times', color='blue')
plt.title("Time vs n (Merge Sort)")
plt.xlabel("n (Input Size)")
plt.ylabel("Time (seconds)")
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(ns, ops,label='Basicop', color='green')
plt.title("Performance of Merge Sort in terms of no of basicop")
plt.xlabel("n (Input Size)")
plt.ylabel("Basic Operations Count")
print("Basic Operation")
print(ops)
n=[i for i in range(10,10000)]
```

```
import math

m=[i*math.log2(i) for i in range(10,10000)]

plt.plot(n,m,label='nlgn Plot')

plt.legend()

plt.tight_layout()

plt.show()
```

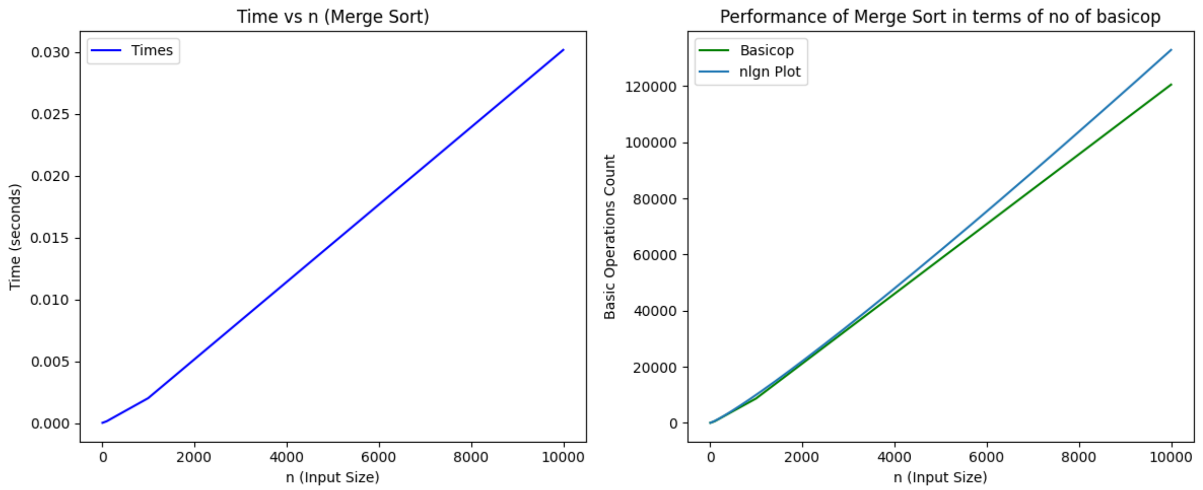
OUTPUT:

Basic Operation

[22, 533, 8685, 120514]

Times:

[2.3365020751953125e-05, 0.0001468658447265625, 0.0020155906677246094, 0.030162811279296875]



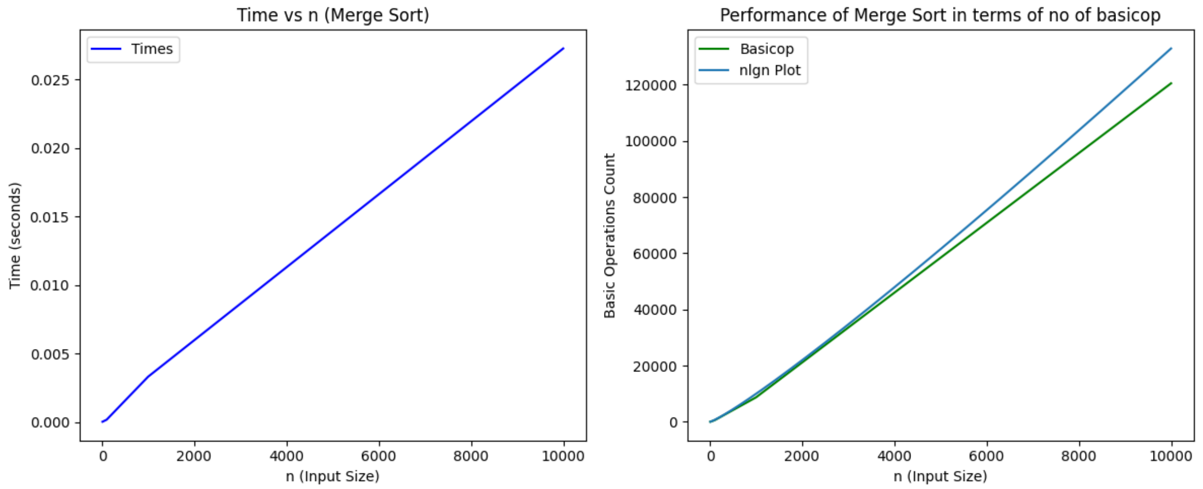
WORST CASE:

Basic Operation

[24, 542, 8697, 120474]

Times:

[3.1948089599609375e-05, 0.0001800060272216797, 0.0033209323883056640, 0.027277708053588867]



RESULT:

Thus the program to implement Merge Sort methods to sort an array of elements and determine the time required to sort and replace the experiment for different values of n, the number of elements in the list to be sorted and a plot a graph for the time taken vs n has been executed and verified successfully.

EX.NO:18 ***IMPLEMENTATION OF QUICK SORT***

26/04/25

AIM:

To implement Quick Sort to sort an array of elements and determine the time required to sort an array of elements and determine the time required to sort. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

PSEUDOCODE:

PROGRAM:

```
import time
import math
import random
import matplotlib.pyplot as plt
def partition(arr,low,high,nbasicop):
 pivot=arr[high]
 i=low-1
 for j in range(low,high):
  nbasicop+=1
  if arr[j]<=pivot:
   i+=1
   arr[i],arr[j]=arr[j],arr[i]
   nbasicop+=1
 arr[i+1],arr[high]=arr[high],arr[i+1]
 nbasicop+=1
 return i+1,nbasicop
def quick_sort(arr,low,high,nbasicop):
 if low<high:
  pi,nbasicop=partition(arr,low,high,nbasicop)
  nbasicop=quick_sort(arr,low,pi-1,nbasicop)
  nbasicop=quick_sort(arr,pi+1,high,nbasicop)
 return nbasicop
def measure_time(arr):
 nbasicop=0
```

```
 start_time=time.time()
 nbasicop=quick_sort(arr,0,len(arr)-1,nbasicop)
 end_time=time.time()
 return end_time-start_time,nbasicop
def build_best_case(arr):
 if not arr:
  return []
 mid=len(arr)//2
 result=[arr[mid]]
 result.extend(build_best_case(arr[:mid]))
 result.extend(build_best_case(arr[mid+1:]))
 return result
ns=list(range(10,1000))
times=[]
basicops=[]
for n in ns:
    arr=random.sample(range(1,1000),n)
    ar=build_best_case(arr[:])
    t,nbasicop=measure_time(ar)
    times.append(t)
    basicops.append(nbasicop)
print("Basicop:",basicops)
print("Times:",times)
n=[i for i in ns]
m=[i*i for i in ns]
```

```
plt.figure(figsize=(12,6))

plt.subplot(1,2,1)

plt.plot(ns,times,color='red')

plt.title("Time vs n (Quick Sort)")

plt.xlabel("n (Input Size)")

plt.ylabel("Time (seconds)")

plt.subplot(1,2,2)

plt.plot(n,m,label="n*n plot")

plt.plot(ns,basicops,label='Basic Operations',color='purple')

plt.title("Quick Sort Performance: Basic Operations")

plt.xlabel("n (Input Size)")

plt.ylabel("Basic Operations Count")

plt.legend()

plt.show()
```

OUTPUT:

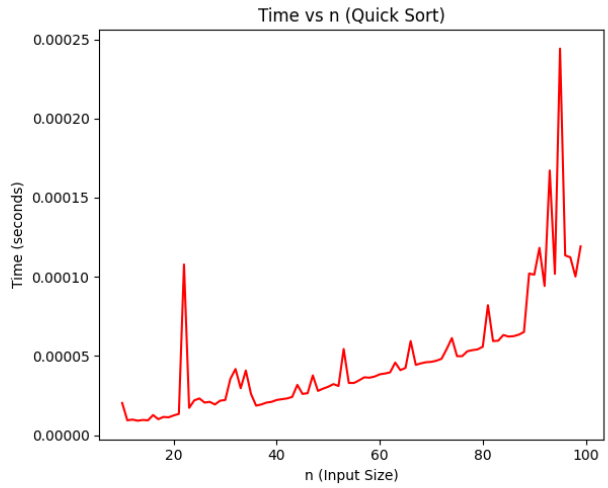
WORST CASE:

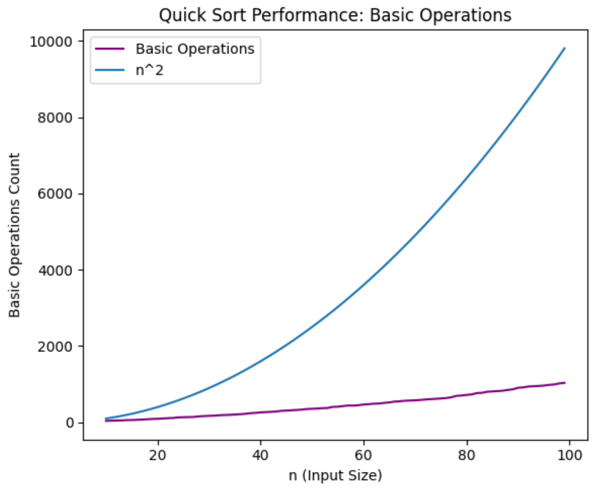
Basic Operations:

[38, 44, 48, 51, 58, 61, 66, 73, 81, 88, 93, 101, 108, 115, 129, 134, 137, 141, 156, 164, 171, 175, 186, 192, 197, 205, 213, 222, 240, 248, 261, 267, 275, 284, 301, 309, 316, 324, 334, 349, 356, 362, 372, 376, 405, 410, 426, 442, 440, 449, 468, 476, 489, 494, 510, 522, 544, 551, 566, 572, 577, 586, 599, 608, 616, 626, 637, 656, 694, 703, 719, 733, 767, 774, 802, 811, 819, 829, 849, 867, 907, 916, 939, 947, 954, 963, 982, 993, 1020, 1036]

Times:

[2.0265579223632812e-05, 9.298324584960938e-06, 9.775161743164062e-06, 9.059906005859375e-06, 9.5367431640625e-06, 9.298324584960938e-06, 1.2636184692382812e-05, 1.0013580322265625e-05, 1.1444091796875e-05, 1.1205673217773438e-05, 1.239776611328125e-05, 1.33514404296875e-05, 0.00010776519775390625, 1.71661376953125e-05, 2.193450927734375e-05, ...]

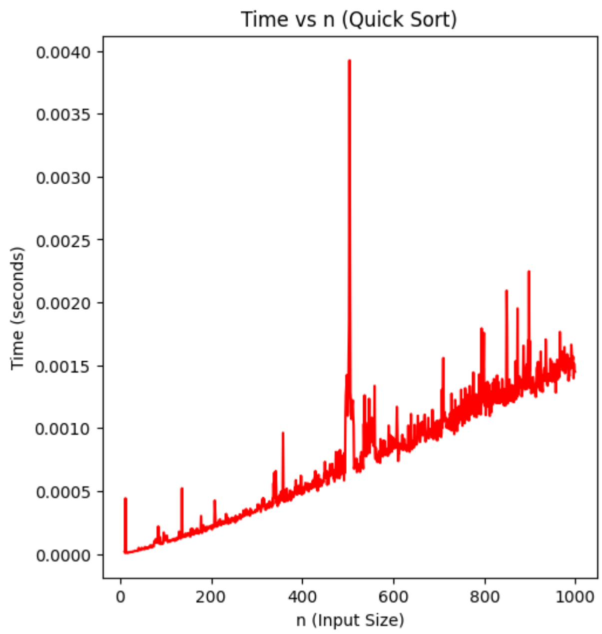


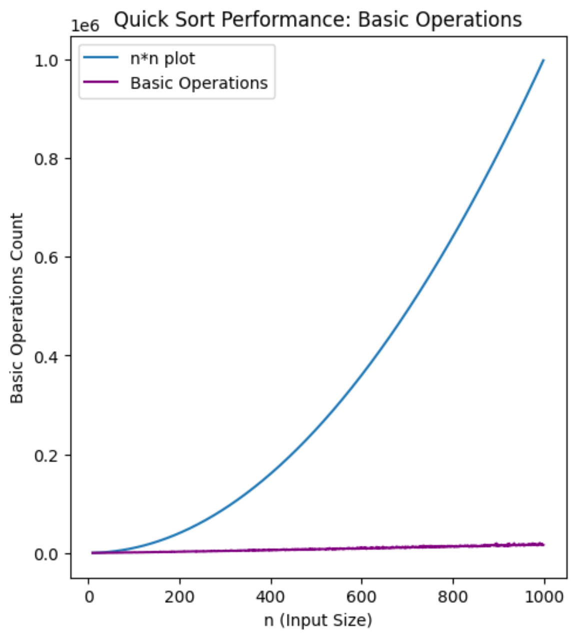


BEST CASE:

Basicop: [43, 66, 47, 58, 69, 80, 70, 84, 109, 105, 116, 134, 191, 129, 137, 150, 139, 207, 166, 184, 176, 212, 235, 292, 240, 227, 229, 247, 253, 250, 424, 446, 329, 296, 333, 308, 327, 390, 372, 418, 407, 474, 397, 426, 361, 415, 456, 512, 457, 536.....]

Times: [1.7881393432617188e-05, 8.344650268554688e-06, 0.0004425048828125, 1.3828277587890625e-05, 9.298324584960938e-06, 1.0013580322265625e-05, 8.344650268554688e-06, 1.2636184692382812e-05, 1.1205673217773438e-05, 1.2159347534179688e-05, 1.2636184692382812e-05....]





RESULT:

Thus the program to implement quick sort methods to sort an array of elements and determine the time taken to sort and report the experiment for different values of n,the number of elements in the list to be sorted and to plot a graph for the time taken vs n has been executed and verified successfully.

EX.NO:19 IMPLENTATION OF N QUEENS PROBLEM USING BACKTRACKING

29/04/25

AIM:

To implement N Queens problem using backtracking.

PSEUDOCODE:

PROGRAM:

```
import time
import matplotlib.pyplot as plt
basic_op = 0
backtrack_count=0
x = []
def Place(k, i):
  global basic_op, x,backtrack_count
  for j in range(1, k):
    basic_op += 1
    if x[j] == i or abs(x[j] - i) == abs(j - k):
      return False
  return True
def NQueens(k, n):
  global x,backtrack_count
  for i in range(1, n + 1):
    if Place(k, i):
      x[k] = i
      if k < n:
        backtrack_count+=1
        NQueens(k + 1, n)
  return basic_op,backtrack_count
def measure_time(n):
  global x, basic_op
  x = [0] * (n + 1)
```

```
    basic_op = 0
    start = time.time()
    NQueens(1, n)
    end = time.time()
    return end - start, basic_op, backtrack_count
ns = list(range(4,12))
times = []
ops = []
for n in ns:
    t, op,bt = measure_time(n)
    times.append(t)
    ops.append(op)
    print("N Value:",n)
    print("Basicop:",op)
    print("Backtrack:",bt)
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(ns, times, marker='o', color='blue')
plt.title('N vs Time Taken')
plt.xlabel('N (Board Size)')
plt.ylabel('Time (seconds)')
plt.subplot(1, 2, 2)
plt.plot(ns, ops,label='basicop', color='green')
plt.title('Performance of N Queens Problem in terms on Number of Basic Operations')
plt.xlabel('N (Board Size)')
```

```
plt.ylabel('Number of Basic Operations')

print("Basic Operation")

print(ops)

n=[i for i in range(4,12)]

import math

m=[math.factorial(i) for i in range(4,12)]

plt.plot(n,m,label='Factorial Plot')

plt.legend()

plt.tight_layout()

plt.show()
```

OUTPUT:

N Value: 4

Basicop: 84

Backtrack: 14

N Value: 5

Basicop: 405

Backtrack: 57

N Value: 6

Basicop: 2016

Backtrack: 205

N Value: 7

Basicop: 9297

Backtrack: 716

N Value: 8

Basicop: 46752

Backtrack: 2680

N Value: 9

Basicop: 243009

Backtrack: 10721

N Value: 10

Basicop: 1297558

Backtrack: 45535

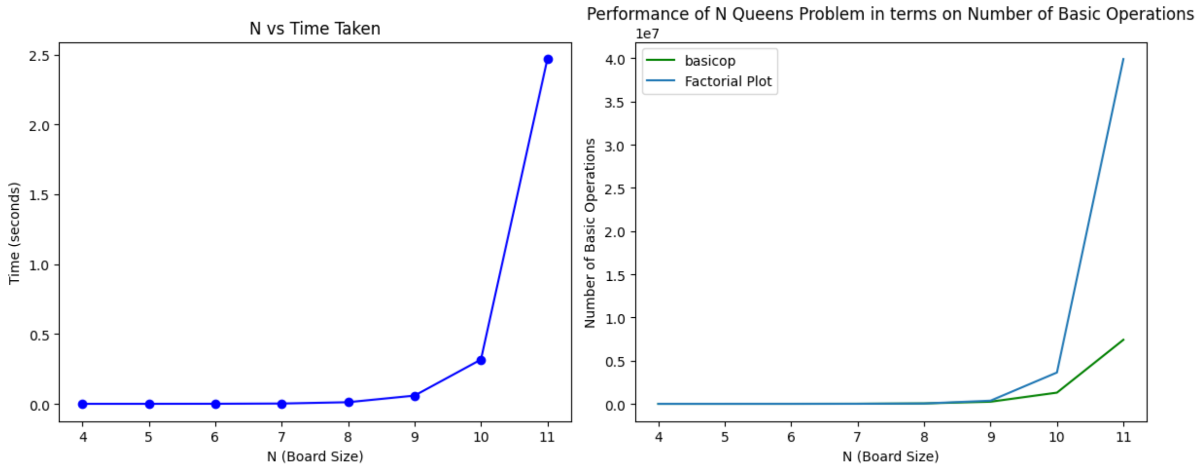
N Value: 11

Basicop: 7416541

Backtrack: 209780

Basic Operation

[84, 405, 2016, 9297, 46752, 243009, 1297558, 7416541]



RESULT:

Thus the program to implement N Queens problem using Backtracking has been executed and verified successfully.

EX.No:20 IMPLEMENTATION OF TRAVELING SALESPERSON

06/05/25 PROBLEM USING APPROXIMATION ALGORITHM

AIM:

To implement any scheme to find the optimal solution for the traveling salesperson problem and to solve the same problem instance using any approximation algorithm and determine the error in the approximation.

PSEUDOCODE:

PROGRAM:

```
import time
import heapq
import random
import math
from itertools import permutations
import matplotlib.pyplot as plt
def measure_time(func, *args, **kwargs):
    start = time.time()
    result = func(*args, **kwargs)
    end = time.time()
    elapsed = end - start
    return result, elapsed
def generate_random_distance_matrix(n, min_w=1, max_w=100):
    matrix = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(i+1, n):
            weight = random.randint(min_w, max_w)
            matrix[i][j] = weight
            matrix[j][i] = weight
    return matrix
def prim_mst(graph):
    n = len(graph)
    in_mst = [False] * n
    parent = [-1] * n
```

```
    key = [float('inf')] * n
    key[0] = 0
    pq = [(0, 0)]  # (key, vertex)
    nbasicop = 0
    while pq:
        k, u = heapq.heappop(pq)
        if in_mst[u]:
            continue
        nbasicop += 1
        in_mst[u] = True
        for v in range(n):
            if not in_mst[v] and graph[u][v] < key[v]:
                key[v] = graph[u][v]
                parent[v] = u
                heapq.heappush(pq, (key[v], v))
    return parent, nbasicop
def parent_to_adjlist(parent):
    n = len(parent)
    adj_list = [[] for _ in range(n)]
    for v in range(1, n):
        u = parent[v]
        adj_list[u].append(v)
        adj_list[v].append(u)
    return adj_list
def preorder_traversal(adj_list, start=0):
```

```
    visited = set()
    tour = []
    def dfs(u):
        visited.add(u)
        tour.append(u)
        for v in adj_list[u]:
            if v not in visited:
                dfs(v)
    dfs(start)
    tour.append(start)  # Return to start
    return tour
def compute_tour_cost(tour, graph):
    cost = 0
    for i in range(len(tour) - 1):
        cost += graph[tour[i]][tour[i + 1]]
    return cost
def exact_tsp_solver(graph):
    n = len(graph)
    if n > 10:
        return None  # Too slow for large n
    min_cost = float('inf')
    best_path = []
    for perm in permutations(range(1, n)):
        tour = [0] + list(perm) + [0]
        cost = compute_tour_cost(tour, graph)
```

```
        if cost < min_cost:
            min_cost = cost
            best_path = tour
    return min_cost, best_path
def tsp_approximation_from_matrix(graph, optimal_cost=None):
    (parent, nbasicop), mst_time = measure_time(prim_mst, graph)
    adj_list = parent_to_adjlist(parent)
    tsp_tour = preorder_traversal(adj_list)
    tsp_cost = compute_tour_cost(tsp_tour, graph)
    approx_error = None
    if optimal_cost:
        approx_error = ((tsp_cost - optimal_cost) / optimal_cost) * 100
    return tsp_cost, approx_error, nbasicop, mst_time, tsp_tour
n_values = list(range(3,10))
random.seed(42)
approximation_costs = []
optimal_costs = []
basicops = []
times = []
approximation_errors = []
for n in n_values:
    print(f"Running for n = {n}")
    random_matrix = generate_random_distance_matrix(n)
    optimal_cost = None
    if n <= 10:
```

```
        (optimal_cost, optimal_tour), exact_time = measure_time(exact_tsp_solver, random_matrix)
    else:
        print("Skipping exact TSP (too large).")
    print(f"Exact TSP Cost: {optimal_cost}")
    tsp_cost, approx_error, nbasicop, mst_time, tsp_tour = tsp_approximation_from_matrix(random_matrix,
optimal_cost)
    print(f"Approximation Path (Tour): {tsp_tour}")
    print(f"Approximation Cost: {tsp_cost}")
    basicops.append(nbasicop)
    times.append(mst_time)
    approximation_costs.append(tsp_cost)
    optimal_costs.append(optimal_cost)
    approximation_errors.append(approx_error)
print(f"\nBasic Operations (Prim's MST): {basicops}")
print("Time Taken (MST + Approx):", times)
print("Approximation cost:", approximation_costs)
print("Approximation error:", approximation_errors)
```

OUTPUT:

Running for n = 3

Exact TSP Cost: 101

Approximation Path (Tour): [0, 2, 1, 0]

Approximation Cost: 101

Running for n = 4

Exact TSP Cost: 115

Approximation Path (Tour): [0, 3, 1, 2, 0]

Approximation Cost: 115

Running for n = 5

Exact TSP Cost: 113

Approximation Path (Tour): [0, 1, 2, 3, 4, 0]

Approximation Cost: 113

Running for n = 6

Exact TSP Cost: 211

Approximation Path (Tour): [0, 1, 3, 4, 2, 5, 0]

Approximation Cost: 211

Running for n = 7

Exact TSP Cost: 133

Approximation Path (Tour): [0, 2, 4, 1, 3, 5, 6, 0]

Approximation Cost: 178

Running for n = 8

Exact TSP Cost: 150

Approximation Path (Tour): [0, 4, 2, 5, 6, 3, 7, 1, 0]

Approximation Cost: 250

Running for n = 9

Exact TSP Cost: 200

Approximation Path (Tour): [0, 1, 3, 6, 5, 8, 4, 2, 7, 0]

Approximation Cost: 293

Basic Operations (Prim's MST): [3, 4, 5, 6, 7, 8, 9]

Time Taken (MST + Approx): [4.029273986816406e-05, 1.1682510375976562e-05, 1.3828277587890625e-05, 1.4066696166992188e-05, 2.1696090698242188e-05, 3.981590270996094e-05, 4.696846008300781e-05]

Approximation cost: [101, 115, 113, 211, 178, 250, 293]

Approximation error: [0.0, 0.0, 0.0, 0.0, 33.83458646616541, 66.66666666666666, 46.5]

RESULT:

Thus the program to implement any scheme to find the optimal solution for the traveling salesperson problem and to solve the same problem instance using any approximation algorithm and determination of error in the approximation has been executed and verified successfully.

EX.NO:21 **IMPLEMENTATION OF RANDOMIZED ALGORITHM FOR**

13/05/25 **FINDING THE Kth SMALLEST NUMBER**

AIM:

To implement randomized algorithms for finding the kth smallest number.

PSEUDOCODE:

PROGRAM:

```
import random

import time

nbasicop = 0

def randomized_partition(arr, low, high):

  global nbasicop

  pivot_index = random.randint(low, high)

  arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

  pivot = arr[high]

  i = low - 1

  for j in range(low, high):

    nbasicop += 1  # Count comparison

    if arr[j] <= pivot:

      i += 1

      arr[i], arr[j] = arr[j], arr[i]

  arr[i+1], arr[high] = arr[high], arr[i+1]

  return i + 1

def randomized_quickselect(arr, low, high, k):

  if low == high:

    return arr[low]

  pivot_index = randomized_partition(arr, low, high)

  count = pivot_index - low + 1

  if k == count:

    return arr[pivot_index]

  elif k < count:
```

```
        return randomized_quickselect(arr, low, pivot_index - 1, k)
    else:
        return randomized_quickselect(arr, pivot_index + 1, high, k - count)
ns = [1000, 2000, 5000, 10000, 20000, 50000, 100000]
times=[]
basicops=[]
for n in ns:
    arr = random.sample(range(1, n*10), n)
    k = random.randint(1, n)
    nbasicop = 0
    start = time.time()
    kth_element = randomized_quickselect(arr.copy(), 0, n-1, k)
    end = time.time()
    elapsed = end - start
    print("n:",n)
    print("Times:",elapsed)
    print("nbasicop:",nbasicop)
    times.append(elapsed)
    basicops.append(nbasicop)
print("Basicop:",basicops)
print("Times:",times)
```

OUTPUT:

```
n: 1000
Times: 0.0005917549133300781
nbasicop: 2523
n: 2000
Times: 0.001584768295288086
nbasicop: 6893
n: 5000
Times: 0.004640340805053711
```

```
nbasicop: 21854
n: 10000
Times: 0.006039619445800781
nbasicop: 26545
n: 20000
Times: 0.011452198028564453
nbasicop: 51630
n: 50000
Times: 0.03369450569152832
nbasicop: 115951
n: 100000
Times: 0.08425402641296387
nbasicop: 218466
Basicop: [2523, 6893, 21854, 26545, 51630, 115951, 218466]
Times: [0.0005917549133300781, 0.001584768295288086, 0.004640340805053711,
0.006039619445800781, 0.011452198028564453, 0.03369450569152832,
0.08425402641296387]
```

RESULT:

Thus the program to implement randomized algorithms for finding the Kth smallest number has been executed and verified successfully.