APPENDIX B

LAB PROGRAMS USING PYTHON

Contents

1. Write a program to implement linear search algorithm Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.
2. Write a program to implement binary search algorithm. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of the time taken versus n.
3. Write a program to solve towers of honai problem and execute it for different number of disks
4. Write a Program to Sort a given set of numbers using selection sort algorithm. 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. The elements can be read from a file or can be generated using the random number generator.
5. Write a program to find the value of an (where a and n are integers) using both brute-force based algorithm and divide and conquer based algorithm
6. Write a Program to Sort a given set of elements using quick sort algorithm. 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.
7. Write a Program to find the binomial co-efficient C(n, k), [where n and k are integers and $n > k$] using brute force based algorithm and also dynamic programming based algorithm
8. Write a Program to implement Floyd's algorithm and find the lengths of the shortest paths from every pairs of vertices in a given weighted graph
9. Write a program to evaluate a polynomial using brute-force based algorithm and using Horner's rule and compare their performances
10. Write a Program to solve the string matching problem using Boyer-Moore approach.
11. Write a Program to solve the string matching problem using KMP algorithm
12. Write a program to implement BFS traversal algorithm
13. Write a program to find the minimum spanning tree of a given graph using Prim's algorithm
14. Write a Program to obtain the topological ordering of vertices in a given digraph. Compute the transitive closure of a given directed graph using Warshall's algorithm.
15. Write a Program to Find a subset of a given set $S = \{s1,s2,,sn\}$ of n positive integers whose ..... sum is equal to a given positive integer d. For example, if $S= \{1, 2, 5, 6, 8\}$ and $d = 9$ there are two solutions $\{1,2,6\}$ and $\{1,8\}$.A suitable message is to be displayed if the given problem instance doesn't have a solution.

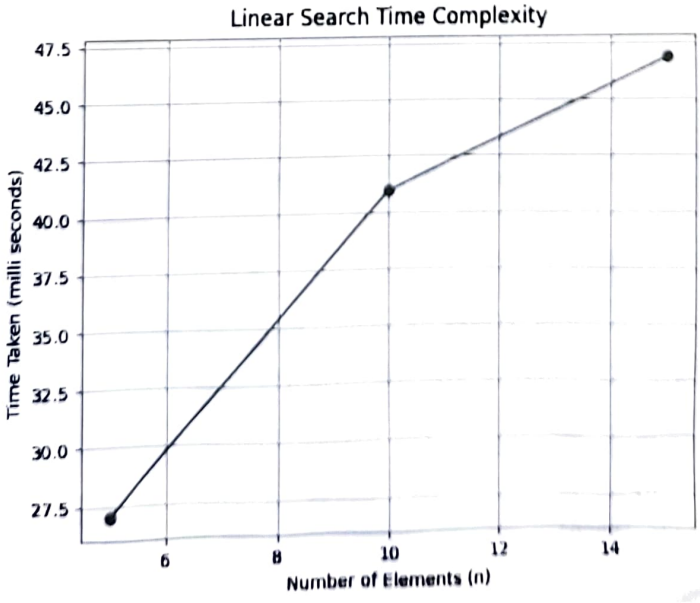
Program 1 Write a program to implement linear search algorithm. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of time taken versus n

```
import time
import matplotlib.pyplot as plt

# Function to perform linear search
def linear_search(arr, n, key):
    for i in range(n):
        if arr[i] == key:
            return i  # If key is found, return the index
    return -1  # If key is not found, return -1

# Function to run linear search multiple times and record results
def linear_search(r):
    results = []
    for _ in range(r):
        n = int(input("Enter the number of elements: "))
        arr = list(map(int, input("\nEnter the elements of an array: ").split()))
        key = int(input("\nEnter the key element to be searched: "))

        # Repeat the search operation multiple times to amplify the time taken
        repeat = 10000
        result = -1

        start = time.time()
        for _ in range(repeat):
            result = linear_search(arr, n, key)
        end = time.time()
        if result != -1:
            print(f"Key {key} found at position {result}")
        else:
            print(f"Key {key} not found")

        time_taken = (end - start) * 1000  # In milli seconds
        print(f"Time taken to search a key element = {time_taken} milli seconds\n")

        # Record number of elements and time taken
        results.append((n, time_taken))
    return results
# Function to plot results
def plot_results(results):
    # Extract data
    n_values = [result[0] for result in results]
    times = [result[1] for result in results]
```

```
    # Create plot
    plt.figure()
    plt.plot(n_values, times, 'o-')
    plt.xlabel('Number of Elements (n)')
    plt.ylabel('Time Taken (milli seconds)')
    plt.title('Linear Search Time Complexity')
    plt.grid(True)
    plt.show()

# Main function
r = int(input("Enter the number of runs: "))
results = linear_search(r)
plot_results(results)
```

Output

```
Enter the number of runs: 3
Enter the number of elements: 15
Enter the elements of an array: 150 140 130 120 110 100 90 80 70 60 50 40 30 20 10

Enter the key element to be searched: 10
Key 10 found at position 14
Time taken to search a key element = 46.87952995300293 milli seconds

Enter the number of elements: 10
Enter the elements of an array: 100 90 80 70 60 50 40 30 20 10
Enter the key element to be searched: 10
Key 10 found at position 9
Time taken to search a key element = 40.95292091369629 milli seconds

Enter the number of elements: 5
Enter the elements of an array: 50 40 30 20 10
Enter the key element to be searched: 10
Key 10 found at position 4
Time taken to search a key element = 26.984214782714844 milli seconds
```



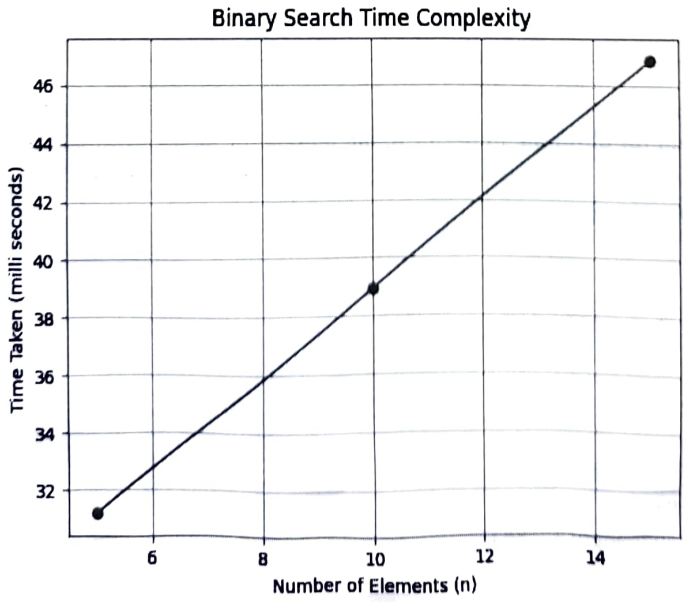
Program 2 Write a program to implement binary search algorithm. Repeat the experiment for different values of n, the number of elements in the list to be searched and plot a graph of time taken versus n.

```
# Function to perform binary search
def binary_search(arr, low, high, key):
    while low <= high:
        mid = low + (high - low) // 2
        if arr[mid] == key:
            return mid  # If key is found, return the index
        if arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1  # If key is not found, return -1
# Main function
def main():
    n_values = []
    times = []

    r = int(input("Enter the number of runs: "))
    for _ in range(r):
        n = int(input("Enter the number of elements: "))
        arr = sorted(list(map(int, input("\nEnter the elements of an array: ").split())))
        key = int(input("\nEnter the key element to be searched: "))
        # Repeat the search operation multiple times to amplify the time taken
        repeat = 10000

        start = time.time()
        for _ in range(repeat):
            result = binary_search(arr, 0, n - 1, key)
        end = time.time()

        if result != -1:
            print(f"Key {key} found at position {result}")
        else:
            print(f"Key {key} not found")

        time_taken = (end - start) * 1000  # In milli seconds
        print(f"Time taken to search a key element = {time_taken} milli seconds\n")

        # Append to the lists for plotting
        n_values.append(n)
        times.append(time_taken)
```

```
    # Plotting
    plt.figure()
    plt.plot(n_values, times, 'o-')
    plt.xlabel('Number of Elements (n)')
    plt.ylabel('Time Taken (milli seconds)')
    plt.title('Binary Search Time Complexity')
    plt.grid(True)
    plt.show()

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

Output

```
Enter the number of runs: 3
Enter the number of elements: 15
Enter the elements of an array: 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150
Enter the key element to be searched: 150
Key 150 found at position 14
Time taken to search a key element = 46.845197677612305 milli seconds

Enter the number of elements: 10
Enter the elements of an array: 10 20 30 40 50 60 70 80 90 100
Enter the key element to be searched: 100
Key 100 found at position 9
Time taken to search a key element = 38.94662857055664 milli seconds

Enter the number of elements: 5
Enter the elements of an array: 10 20 30 40 50
Enter the key element to be searched: 50
Key 50 found at position 4
Time taken to search a key element = 31.207799911499023 milli seconds
```



Program 3 Write a program to solve towers of Hanoi problem and execute it for different number of disks.

```
def toh(n, source, temp, dest):
    global count
    if n > 0:
        toh(n-1, source, dest, temp)
        print(f"Move Disk {n} {source}->{dest}")
        count += 1
        toh(n-1, temp, source, dest)

# Main Code
source = 'S'
temp = 'T'
dest = 'D'
count = 0

n = int(input("Enter the number of disks: "))
print("Sequence is:")
toh(n, source, temp, dest)
print("The Number of Moves:", count)
```

Output

Run 1:

```
Enter the number of disks: 2
Sequence is:
Move Disk 1 S->T
Move Disk 2 S->D
Move Disk 1 T->D
The Number of Moves: 3
```

Run 2:

```
Enter the number of disks: 3
Sequence is:
Move Disk 1 S->D
Move Disk 2 S->T
Move Disk 1 D->T
Move Disk 3 S->D
Move Disk 1 T->S
Move Disk 2 T->D
Move Disk 1 S->D
The Number of Moves: 7
```

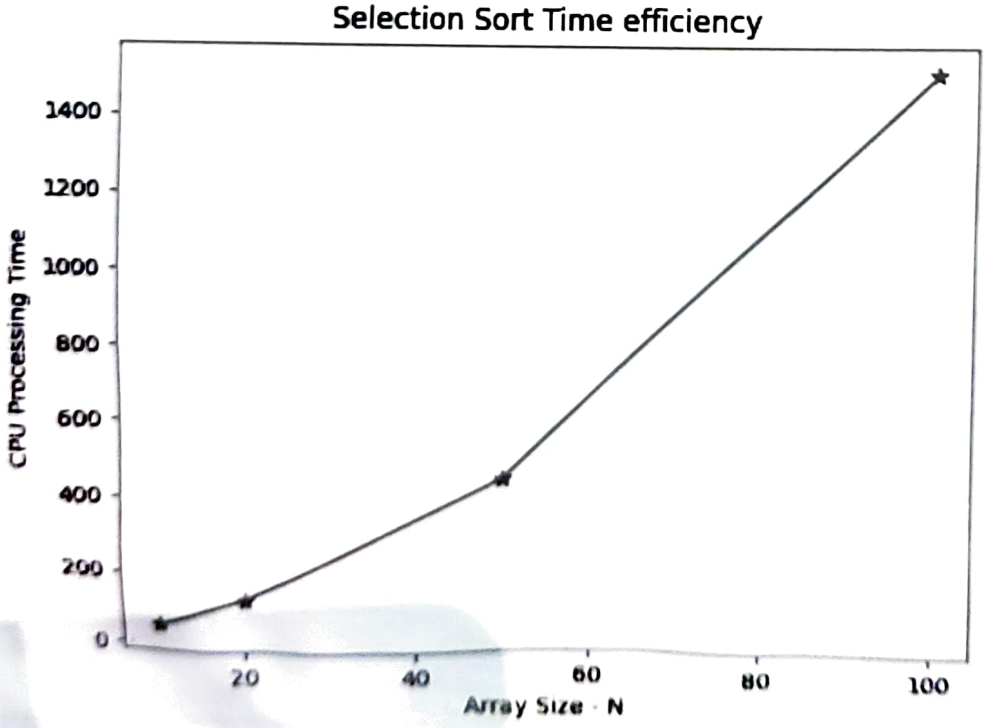
Program 4 Write a program to sort a given set of numbers using selection sort algorithm. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n. The elements can be read from a file or can be generated using random number generator.

```
import timeit
import random
import matplotlib.pyplot as plt

# Input Array elements
def Input(Array, n):
    # iterating till the range
    for i in range(0, n):
        ele = random.randrange(1, 50)
        # adding the element
        Array.append(ele)
```

```
# Selection Sort
def selectionSort(Array, size):
    for ind in range(size):
        min_index = ind
        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if Array[j] < Array[min_index]:
                min_index = j
        # swapping the elements to sort the array
        (Array[ind], Array[min_index]) = (Array[min_index], Array[ind])
# Main Block
N = []
CPU = []
trail = int(input("Enter no. of trails: "))
for t in range(0, trail):
    Array = []
    print("-----> TRAIL NO:", t + 1)
    n = int(input("Enter number of elements: "))
    Input(Array, n)
    start = timeit.default_timer()
    selectionSort(Array, n)
    times = timeit.default_timer() - start
    print("Sorted Array:")
    print(Array)
    N.append(n)
    CPU.append(round(float(times) * 1000000, 2))

print("N CPU")
for t in range(0, trail):
    print(N[t], CPU[t])

# Plotting Graph
plt.plot(N, CPU)
plt.scatter(N, CPU, color="red", marker="*", s=50)
# naming the x axis
plt.xlabel('Array Size - N')
# naming the y axis
plt.ylabel('CPU Processing Time')
# giving a title to graph
plt.title('Selection Sort Time efficiency')
# function to show the plot
plt.show()
```

Output

```
Enter no. of trails: 4
-----> TRAIL NO: 1
Enter number of elements: 10
Sorted Array:
[6, 7, 15, 17, 27, 29, 32, 36, 39, 46]
-----> TRAIL NO: 2
Enter number of elements: 20
Sorted Array:
[6, 8, 8, 11, 13, 13, 14, 18, 19, 20, 21, 28, 29, 32, 33, 35, 36, 44, 44, 44]
-----> TRAIL NO: 3
Enter number of elements: 50
Sorted Array:
[2, 3, 5, 5, 6, 6, 6, 7, 8, 9, 9, 10, 10, 11, 14, 15, 17, 17, 20, 20, 20, 21, 22, 22, 24, 25,
26, 27, 28, 28, 29, 32, 33, 36, 38, 38, 39, 40, 40, 41, 41, 41, 42, 43, 43, 44, 44, 46, 49, 49]
-----> TRAIL NO: 4
Enter number of elements: 100
Sorted Array:
[1, 1, 2, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 9, 9, 9, 12, 12, 13, 13, 14, 15, 15, 16, 16, 16, 18, 18,
19, 20, 20, 20, 21, 21, 21, 21, 22, 22, 23, 23, 24, 24, 24, 24, 24, 25, 25, 25, 26, 26, 26, 27,
27, 30, 30, 31, 31, 32, 32, 33, 34, 34, 34, 34, 35, 36, 36, 36, 36, 37, 37, 38, 39, 39, 39, 39,
39, 40, 40, 40, 41, 41, 41, 42, 42, 42, 43, 44, 44, 44, 44, 45, 45, 46, 46, 47, 47, 48, 49, 49]
N CPU
10 54.7
20 117.5
50 446.2
100 1509.9
```



Program 5 Write a program to find a^n using (a) Brute-force based algorithm (b) Divide and conquer based algorithm

```
def power_bruteforce(a, n):
    result = 1
    for i in range(n):
        result *= a
    return result

def power_divide_conquer(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return power_divide_conquer(a * a, n // 2)
    else:
        return a * power_divide_conquer(a * a, n // 2)

# Main Code
a, n = map(int, input("Enter the value of a and n: ").split())

result_brute = power_bruteforce(a, n)
result_divide_conquer = power_divide_conquer(a, n)

print("Result using brute force:", result_brute)
print("Result using divide and conquer:", result_divide_conquer)
```

Output

```
Enter the value of a and n: 2 8
Result using brute force: 256
Result using divide and conquer: 256
```

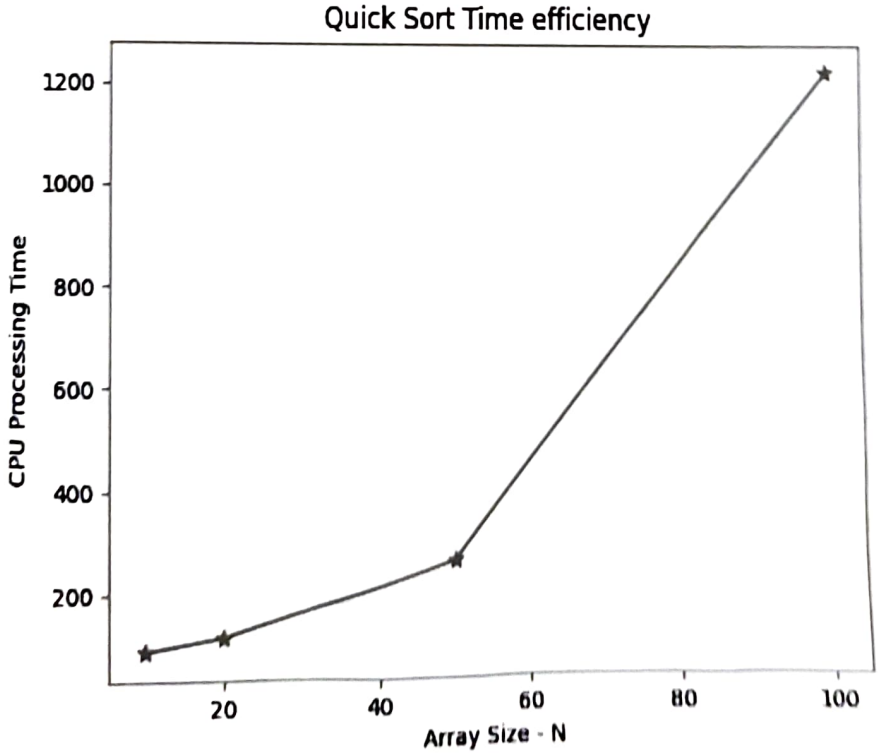
Program 6 Write a program to sort a given set of numbers using quick sort algorithm. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

```
import timeit
import random
import matplotlib.pyplot as plt

# Input Array elements
def Input(Array, n):
    # iterating till the range
    for i in range(0, n):
        ele = random.randrange(1, 50)
        # adding the element
        Array.append(ele)
# divide function
def partition(Array, low, high):
    i = (low - 1)
    pivot = Array[high]  # pivot element
```

```
    for j in range(low, high):
        # If current element is smaller
        if Array[j] <= pivot:
            # increment
            i = i + 1
            Array[i], Array[j] = Array[j], Array[i]
    Array[i + 1], Array[high] = Array[high], Array[i + 1]
    return (i + 1)
# Quick sort
def quickSort(Array, low, high):
    if low < high:
        # index
        pi = partition(Array, low, high)
        # sort the partitions
        quickSort(Array, low, pi - 1)
        quickSort(Array, pi + 1, high)

# Main Block
N = []
CPU = []
trail = int(input("Enter no. of trails: "))
for t in range(0, trail):
    Array = []
    print("-----> TRAIL NO:", t + 1)
    n = int(input("Enter number of elements: "))
    Input(Array, n)
    start = timeit.default_timer()
    quickSort(Array, 0, n - 1)
    times = timeit.default_timer() - start
    print("Sorted Array:")
    print(Array)
    N.append(n)
    CPU.append(round(float(times) * 1000000, 2))

print("N CPU")
for t in range(0, trail):
    print(N[t], CPU[t])

# Plotting Graph
plt.plot(N, CPU)
plt.scatter(N, CPU, color="red", marker="*", s=50)
# naming the x axis
plt.xlabel('Array Size - N')
# naming the y axis
plt.ylabel('CPU Processing Time')
# giving a title to graph
plt.title('Quick Sort Time efficiency')
# function to show the plot
plt.show()
```

Output

```
Enter no. of trails: 4
-----> TRAIL NO: 1
Enter number of elements: 10
Sorted Array:
[1, 3, 8, 29, 34, 39, 40, 42, 45, 46]
-----> TRAIL NO: 2
Enter number of elements: 20
Sorted Array:
[4, 7, 8, 8, 9, 10, 15, 17, 18, 20, 20, 23, 23, 24, 32, 36, 42, 42, 44, 45]
-----> TRAIL NO: 3
Enter number of elements: 50
Sorted Array:
[2, 4, 7, 7, 8, 9, 10, 10, 11, 11, 13, 13, 15, 15, 16, 18, 18, 19, 20, 21, 23, 24, 25, 26, 27, 28,
30, 30, 30, 32, 33, 33, 34, 34, 37, 38, 39, 39, 41, 44, 45, 45, 46, 46, 46, 47, 47, 49, 49, 49]
-----> TRAIL NO: 4
Enter number of elements: 100
Sorted Array:
[1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 8, 9, 9, 11, 11, 11, 13, 14,
14, 15, 16, 16, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 20, 20, 21, 23, 23, 23, 23, 24, 24, 26,
26, 26, 26, 27, 28, 28, 28, 29, 29, 31, 31, 31, 32, 33, 34, 34, 36, 37, 37, 37, 38, 39, 39, 41,
42, 42, 42, 42, 42, 42, 42, 43, 43, 43, 43, 44, 44, 45, 47, 47, 47, 47, 48, 48, 48, 49]
N CPU
10 89.7
20 114.2
50 259.6
100 1225.0
```



Program 7 **Write a program to find binomial co-efficient C(n,k) [where n and k are integers and n > k] using brute force algorithm and also dynamic programming based algorithm.**

```
# Function to calculate factorial for brute force method
def factorial(n):
    fact = 1
    for i in range(2, n + 1):
        fact *= i
    return fact

# Brute force method to find binomial coefficient
def binomialCoeff_bruteForce(n, k):
    return factorial(n) // (factorial(k) * factorial(n - k))

# Dynamic programming method to find binomial coefficient
def binomialCoeff_DP(n, k):
    C = [[0 for j in range(k + 1)] for i in range(n + 1)]
    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            # Base Cases
            if j == 0 or j == i:
                C[i][j] = 1
            # Calculate value using previously stored values
            else:
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j]
    return C[n][k]

# Main Code
n = int(input("Enter the value of n: "))
k = int(input("Enter the value of k: "))
result_bruteForce = binomialCoeff_bruteForce(n, k)
result_DP = binomialCoeff_DP(n, k)
print(f"Binomial Coefficient (Brute Force): {result_bruteForce}")
print(f"Binomial Coefficient (Dynamic Programming): {result_DP}")
```

Output

```
Enter the value of n: 5
Enter the value of k: 2
Binomial Coefficient (Brute Force): 10
Binomial Coefficient (Dynamic Programming): 10
```

Program 8 Write a program to implement Floyd's algorithm and find the lengths of the shortest paths from every pairs of vertices in a weighted graph.

```
INF = 99999

# Print the solution matrix
def printSolution(V, D):
    print("The following matrix shows the shortest distances between every pair of vertices")
    for i in range(V):
        for j in range(V):
            if D[i][j] == INF:
                print("%7s" % "INF", end="")
            else:
                print("%7d" % D[i][j], end="")
        print()

# Implementing Floyd Warshall Algorithm
def floyd(V, C):
    D = [[0]*V for _ in range(V)]

    for i in range(V):
        for j in range(V):
            D[i][j] = C[i][j]

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if D[i][j] > (D[i][k] + D[k][j]):
                    D[i][j] = D[i][k] + D[k][j]

    printSolution(V, D)

# Main Code
V = int(input("Enter the number of vertices: "))

# allocate memory for the cost matrix
C = [[0]*V for _ in range(V)]

print("Enter the cost matrix row by row (space-separated):")
print("[Enter 99999 for Infinity]")
print("[Enter 0 for cost(i,i)]")
for i in range(V):
    C[i] = list(map(int, input().split()))

floyd(V, C)
```

Output | **Note : Refer Chapter 7. P.No 7.34, Let us considert Example 1 for Inputs.**

```
Enter the number of vertices: 4
Enter the cost matrix row by row (space-separated):
[Enter 99999 for Infinity]
[Enter 0 for cost(i,i)]
0 99999 2 99999
3 0 99999 99999
99999 5 0 1
6 99999 99999 0
The following matrix shows the shortest distances between every pair of vertices
      0      7      2      3
      3      0      5      6
      7      5      0      1
      6     13      8      0
```

Program 9 | **Write a program to evaluate polynomial using brute-force algorithm and using Horner's rule and compare their performances**

```
import time
import math

def bruteForce(coef, n, x):
    sum = 0.0
    for i in range(n+1):
        sum += coef[i] * math.pow(x, i)
    return sum

def hornersRule(coef, n, x):
    result = coef[n]
    for i in range(n-1, -1, -1):
        result = result * x + coef[i]
    return result

# Main Code
n = int(input("Enter the degree of the polynomial: "))

coef = [0]*(n+1)
print("Enter the coefficients from highest degree to lowest:")
for i in range(n, -1, -1):
    coef[i] = int(input())

x = float(input("Enter the value of x: "))
start = time.time()
brute_force_result = bruteForce(coef, n, x)
end = time.time()
time_used = end - start
print(f"Brute force result: {brute_force_result:.2f}, time used: {time_used:.6f} seconds")
```

```
start = time.time()
horners_rule_result = hornersRule(coef, n, x)
end = time.time()
time_used = end - start
print(f"Horner's rule result: {horners_rule_result:.2f}, time used: {time_used:.6f} seconds")
```

Output

```
[Let us evaluate value of 2x³ - 6x² + 2x - 1 for x = 3, co-efficients are {2, -6, 2, -1}]
Enter the degree of the polynomial: 3
Enter the coefficients from highest degree to lowest:
2
-6
2
-1
Enter the value of x: 3
Brute force result: 5.00, time used: 0.000000 seconds
Horner's rule result: 5.00, time used: 0.000000 seconds
```

Program 10 **Write a program to solve the string matching problem using Boyer-Moore approach.**

```
MAX_CHARS = 256

def max(a, b):
    return a if a > b else b

def badCharHeuristic(pat, size, badchar):
    for i in range(MAX_CHARS):
        badchar[i] = -1
    for i in range(size):
        badchar[ord(pat[i])] = i

def patternsearch(text, pat):
    m = len(pat)
    n = len(text)
    badchar = [-1] * MAX_CHARS

    badCharHeuristic(pat, m, badchar)

    s = 0  # s is shift of the pattern with respect to text
    while s <= (n - m):
        j = m - 1

        while j >= 0 and pat[j] == text[s + j]:
            j -= 1

        if j < 0:
            print("\nPattern occurs at position =", s)
            s += m - badchar[ord(text[s + m])] if (s + m) < n else 1
        else:
            s += max(1, j - badchar[ord(text[s + j])])
```

```
# Main Code
text = input("Enter the text: ").rstrip('\n')
pat = input("Enter the pattern: ").rstrip('\n')

patternsearch(text, pat)
```

Output

```
Run 1:
Enter the text: SKYWARDPUBLISHERS
Enter the pattern: PUB
Pattern occurs at position = 7

Run 2:
Enter the text: MALAYALAM
Enter the pattern: LA
Pattern occurs at position = 2
Pattern occurs at position = 6
```

Program 11 Write a program to solve the string matching problem using KMP algorithm.

```
def computeLPSArray(pat, M, lps):
    length = 0
    lps[0] = 0
    i = 1
    while i < M:
        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

def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)
    lps = [0]*M
    computeLPSArray(pat, M, lps)
    i = j = 0
    while i < N:
        if pat[j] == txt[i]:
            i += 1
            j += 1
        if j == M:
            print(f"Found pattern at index {i - j}")
            j = lps[j - 1]
```

```
        elif i < N and pat[j] != txt[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1


# Main Code
txt = input("Enter the text: ")
pat = input("Enter the pattern: ")
KMPSearch(pat, txt)
```

Output

```
Run 1:
Enter the text: SKYWARDPUBLISHERS
Enter the pattern: PUB
Found pattern at index 7

Run 2:
Enter the text: MALAYALAM
Enter the pattern: LA
Found pattern at index 2
Found pattern at index 6
```

Program 12 **Write a program to implement BFS traversal algorithm.**

```
MAX = 100

c = [[0]*MAX for _ in range(MAX)]
visited = [0]*MAX
queue = [0]*MAX

def BFS(v):
    front = 0
    rear = -1

    visited[v] = 1
    queue[rear + 1] = v
    rear += 1

    while front <= rear:
        v = queue[front]
        front += 1

        print(f"{v} ", end="")

        for i in range(1, n+1):
            if c[v][i] == 1 and visited[i] == 0:
                queue[rear + 1] = i
                rear += 1
                visited[i] = 1
```

```
if __name__ == "__main__":
    print("Enter the number of vertices in the graph: ")
    n = int(input())

    print("Enter the cost matrix of the graph: ")
    for i in range(1, n+1):
        c[i] = [0] + list(map(int, input().split()))

    for i in range(1, n+1):
        visited[i] = 0

    print("Enter the starting vertex: ")
    v = int(input())

    print("BFS traversal of the graph is: ", end="")
    BFS(v)
```

Output

Run 1:

[**Note** : We will consider the graph Given in Chapter 2, Section 3.3.3 , Example 1. The adjacency matrix for that graph will be given as an input to this program to get the BFS Traversal of the Graph]

```
Enter the number of vertices in the graph:
6
Enter the cost matrix of the graph:
0 1 1 0 0 0
1 0 0 1 1 0
1 0 0 0 0 1
0 1 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
Enter the starting vertex:
1
BFS traversal of the graph is: 1 2 3 4 5 6
```

Run 2:

[**Note** : We will consider the graph Given in Chapter 2, Section 3.3.3 , Example 2. The adjacency matrix for that graph will be given as an input to this program to get the BFS Traversal of the Graph]

```
Enter the number of vertices in the graph:
9
Enter the cost matrix of the graph:
0 1 0 1 1 0 0 0 0
1 0 1 0 1 0 0 0 0
0 1 0 0 0 1 0 0 0
1 0 0 0 0 0 1 0 0
```

```
1 1 0 0 0 1 1 1 0
0 0 1 0 1 0 0 0 0
0 0 0 1 1 0 0 1 0
0 0 0 0 1 0 1 0 1
0 0 0 0 0 0 0 1 0
Enter the starting vertex:
1
BFS traversal of the graph is: 1 2 4 5 3 7 6 8 9
```

Program 13 **Write a program to find minimum spanning tree of a given graph using Prim's Algorithm.**

```
import sys

def minKey(key, mstSet, n):
    min_value = sys.maxsize
    for v in range(n):
        if mstSet[v] == False and key[v] < min_value:
            min_value = key[v]
            min_index = v
    return min_index

def printMST(parent, c, n):
    totalWeight = 0
    print("Edge   Weight")
    for i in range(1, n):
        print(str(parent[i]+1) + " - " + str(i+1) + "    " + str(c[i][parent[i]]))
        totalWeight += c[i][parent[i]]
    return totalWeight

def primMST(c, n):
    parent = [None]*n
    key = [sys.maxsize]*n
    mstSet = [False]*n
    key[0] = 0
    parent[0] = -1
    for count in range(n):
        u = minKey(key, mstSet, n)
        mstSet[u] = True
        for v in range(n):
            if c[u][v] > 0 and mstSet[v] == False and c[u][v] < key[v]:
                parent[v] = u
                key[v] = c[u][v]
    totalWeight = printMST(parent, c, n)
    print("Total cost of the minimum spanning tree: " + str(totalWeight))
```

```
# Main Code
n = int(input("Enter the number of vertices: "))
c = []
print("Enter the cost adjacency matrix:")
for i in range(n):
    c.append(list(map(int, input().split())))
primMST(c, n)
```

Output

```
[Note : We will consider the undirected weighted graph Given in Chapter 8, Section 8.2.1 ,
Example 1. The adjacency matrix for that graph will be given as an input to this program to get
the minimum cost spanning tree]

Enter the number of vertices: 5
Enter the cost adjacency matrix:
0 11 9 7 8
11 0 15 14 13
9 15 0 12 14
7 14 12 0 6
8 13 14 6 0
Edge   Weight
1 - 2    11
1 - 3    9
1 - 4    7
4 - 5    6
Total cost of the minimum spanning tree: 33
```

Program 14 (a) **Write a program to obtain the topological ordering of vertices in a given digraph.**

```
def main():
    n = int(input("Enter the number of vertices: "))
    count = 0
    c = [[0 for _ in range(n)] for _ in range(n)]
    indeg = [0] * n
    flag = [0] * n
    i, j, k = 0, 0, 0

    print("Enter the cost matrix (row by row):")
    for i in range(n):
        row = input().split()
        for j in range(n):
            c[i][j] = int(row[j])

    for i in range(n):
        for j in range(n):
            indeg[i] += c[j][i]
```

```
    print("The topological order is:")
    while count < n:
        for k in range(n):
            if indeg[k] == 0 and flag[k] == 0:
                print(f"{k+1:3}", end="")
                flag[k] = 1
                count += 1
                for i in range(n):
                    if c[k][i] == 1:
                        indeg[i] -= 1

    return 0

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

Output

[**Note :** We will consider the directed acyclic graph given in Chapter 4, Page No 4.9, Example 1. The adjacency cost matrix for that graph will be given as an input to this program to get the topological ordering of vertices]

Run 1:

```
Enter the number of vertices: 5
Enter the cost matrix (row by row):
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
The topological order is:
  1  2  3  4  5
```

[**Note :** We will consider the directed acyclic graph given in Chapter 4, Page No 4.13, Example 2. The adjacency cost matrix for that graph will be given as an input to this program to get the topological ordering of vertices]

```
Enter the number of vertices: 7
Enter the cost matrix (row by row):
0 0 1 0 0 0 0
0 0 0 1 0 1 0
0 0 0 0 0 0 0
0 0 1 0 0 0 0
0 0 1 0 0 0 0
0 0 0 0 0 0 0
1 0 0 0 0 1 0
The topological order is:
  2  4  5  7  1  3  6
```

Note : Every directed acyclic graph may have one or more topological orderings.

Program 14 (b) Write a program to compute transitive closure of a given directed graph using Warshall's algorithm.

```
def warshalls(c, n):
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if c[i][j] or (c[i][k] and c[k][j]):
                    c[i][j] = 1

    print("The transitive closure of the graph is:")
    for i in range(n):
        for j in range(n):
            print(c[i][j], end=" ")
        print()

def main():
    n = int(input("Enter the number of vertices: "))
    c = []
    print("Enter the adjacency cost matrix:")
    for i in range(n):
        row = list(map(int, input().split()))
        c.append(row)
    warshalls(c, n)

main()
```

Output

[**Note** : We will consider the directed graph given in Chapter 7, Page No 7.26, The adjacency cost matrix for that graph will be given as an input to this program to get the Transitive closure of Directed graph]

```
Enter the number of vertices: 4
Enter the adjacency cost matrix:
0 1 0 0
0 0 0 1
0 0 0 0
1 0 1 0
The transitive closure of the graph is:
1 1 1 1
1 1 1 1
0 0 0 0
1 1 1 1
```

Program 15 **Write a program to find subset of a given set S={s1,s2,....sn} of n positive integers whose sum is equal to given positive integer d. For example if S={1,2,5,6,8} and d=9 then two solutions {1,2,6} and {1,8}. A suitable message is to be displayed if given problem doesn't have solution.**

```
def sum_of_subsets(s, k, r):
    global count, x, w, d, i
    x[k] = 1
    if s + w[k] == d:
        print("\nSubset %d = " % (count + 1), end="")
        for i in range(k + 1):
            if x[i]:
                print("%d " % w[i], end="")
    elif s + w[k] + w[k + 1] <= d:
        sum_of_subsets(s + w[k], k + 1, r - w[k])
    if s + r - w[k] >= d and s + w[k + 1] <= d:
        x[k] = 0
        sum_of_subsets(s, k + 1, r - w[k])

if __name__ == "__main__":
    w = [0] * 10
    x = [0] * 10
    count = 0
    i = 0

    n = int(input("Enter the number of elements: "))
    print("Enter the elements in ascending order: ")
    for i in range(n):
        w[i] = int(input())

    d = int(input("Enter the sum: "))

    sum = 0
    for i in range(n):
        x[i] = 0
        sum += w[i]

    if sum < d or w[0] > d:
        print("\nNo subset possible\n")
    else:
        sum_of_subsets(0, 0, sum)
```

Output

```
Run 1:
Enter the number of elements: 4
Enter the elements in ascending order:
7
11
13
24
Enter the sum: 31

Subset 1 = 7 11 13
Subset 1 = 7 24
Run 2:
Enter the number of elements: 4
Enter the elements in ascending order:
10
20
30
40
Enter the sum: 120

No subset possible
```