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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to perform linear search
int linear_search(int *arr, int n, int key) {
    int i;
    for ( i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i; // If key is found, return the index
        }
    }
    return -1; // If key is not found, return -1
}

int main() {
    int n, i, key;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int *arr = malloc(n * sizeof(int));
    printf("\n Enter the elements of an array : ");
    for(i=0;i<n;i++)
      scanf("%d",&arr[i]);
    printf("\n Enter the key element to be searched : ");
    scanf("%d",&key);

    // Repeat the search operation multiple times to amplify the time taken
    int repeat = 1000000;
    int result;

    clock_t start = clock();
    for (i = 0; i < repeat; i++) {
        result = linear_search(arr, n, key);
    }
    clock_t end = clock();

    if(result != -1) {
        printf(" Key %d Found at Position %d ", key, result);
    } else {
        printf(" Key %d Not Found ", key);
    }

    double time_taken = ((double)end - start) / CLOCKS_PER_SEC * 1000; // In milli seconds
    printf("\n Time taken to search a key element = %f milliseconds\n", time_taken);
    return 0;
}
```

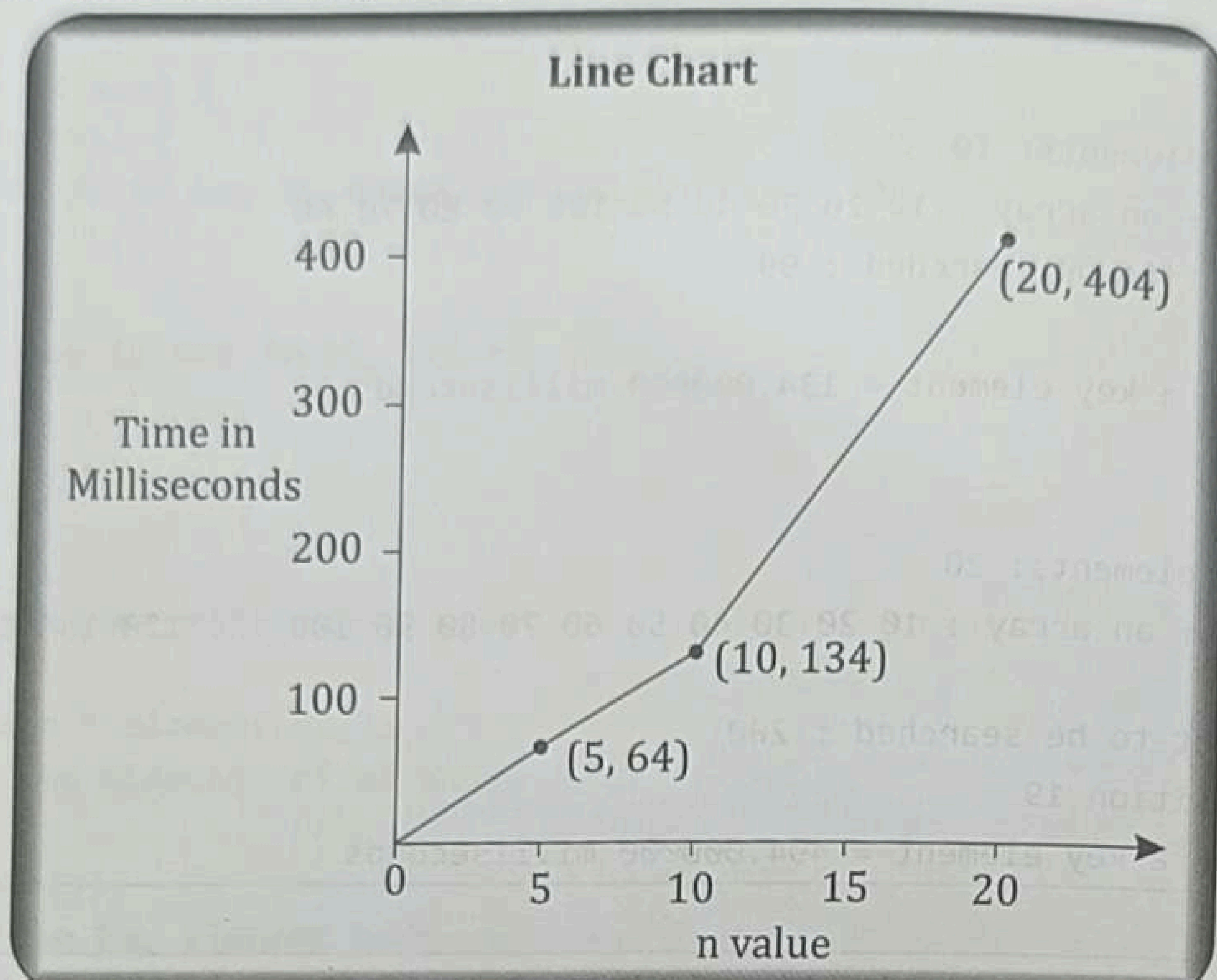
Output

```
Run 1:
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 4
Time taken to search an element = 64.000000 milli seconds

Run 2:
Enter the number of elements: 10
Enter the elements of an array : 10 20 30 40 50 100 90 80 70 60
Enter the key element to be searched : 99
Key 99 Not Found
Time taken to search a key element = 134.000000 milliseconds

Run 3:
Enter the number of elements: 20
Enter the elements of an array : 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180
190 200
Enter the key element to be searched : 200
Key 200 Found at Position 19
Time taken to search a key element = 404.000000 milliseconds
```

Explanation

- The above program is designed to implement the linear search algorithm. This algorithm is one of the simplest search algorithms, where each element in an array is compared with the key element sequentially until the match is found or all elements have been checked.
- The time complexity of the linear search algorithm is O(n), where n is the size of the input. This means that in the worst-case scenario, the algorithm might have to check each and every element in the array once. Thus, the time taken by the linear search algorithm increases linearly with the size of the input.
- The `clock()` function from the time.h library is used to measure the time taken by the linear search operation. The program records the processor time before and after the repeated search operation and calculates the difference to determine the time taken.
- The search operation is repeated multiple times (specified by the `repeat` variable) to amplify the time taken. This helps in measuring the time more accurately, especially when the size of the input (n) is small.
- The program creates an array of size n dynamically using the `malloc` function. This allows the user to specify the size of the array at runtime.
- After performing the search operation, the program prints whether the key was found and its position, or if the key was not found. It also prints the time taken to perform the search operation.
- On testing this program for different values of n, it's observed that the time taken increases linearly. This confirms the linear time complexity of the algorithm. For example, for n=5, it took 64 milliseconds; for n=10, it took 134 milliseconds; and for n=20, it took 404 milliseconds.

- The relationship between n and time taken can be visualized on a line graph with n on the x-axis and time in milliseconds on the y-axis. This will show a line that ascends as n increases, illustrating the linear time complexity of the linear search algorithm. Plotting this graph can help in better understanding and interpreting the performance of the algorithm.
- This can be visualized on a line graph with n values on the x-axis (0, 5, 10, 20) and time in milliseconds on the y-axis (0, 100, 200, 400, 800). We can plot the points (5,64), (10,134), and (20,404) on this graph and draw a line connecting these points as shown below.



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.**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to perform binary search
int binary_search(int *arr, int low, int high, int key)
{
    while(low <= high) {
        int 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
}
```

```
int main()
{
    int n, i, key;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int *arr = malloc(n * sizeof(int));
    printf("\n Enter the elements of an array in ascending order : ");
    for(i=0;i<n;i++)
      scanf("%d",&arr[i]);
    printf("\n Enter the key element to be searched : ");
    scanf("%d",&key);

    // Repeat the search operation multiple times to amplify the time taken
    int repeat = 1000000;
    int result;

    clock_t start = clock();
    for (i = 0; i < repeat; i++) {
        result = binary_search(arr, 0, n - 1, key);
    }
    clock_t end = clock();

    if(result != -1) {
        printf(" Key %d Found at Position %d ", key, result);
    } else {
        printf(" Key %d Not Found ", key);
    }

    double time_taken = ((double)end - start) / CLOCKS_PER_SEC * 1000; // In milli seconds
    printf("\n Time taken to search a key element = %f milliseconds\n", time_taken);
    return 0;
}
```

Output

```
Run 1:
Enter the number of elements: 5
Enter the elements of an array in ascending order : 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 = 83.000000 milliseconds

Run 2:
Enter the number of elements: 10
Enter the elements of an array in ascending order : 10 20 30 40 50 60 70 80 90 100
Enter the key element to be searched : 120
Key 120 Not Found
Time taken to search a key element = 127.000000 milliseconds
```

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.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

int min (int a[ ], int k, int n)
{
    int loc, j, min;
    min = a[k];
    loc = k;
    for (j = k + 1; j <= n - 1; j++)
```

```
        if (min > a [j])
        {
            min = a[j];
            loc = j;
        }
    return (loc);
}

int main()
{
    int i,*arr, k,n,loc=0,temp;
    srand(time(0)); // Seed the random number generator

    printf("Enter the number of elements : ");
    scanf("%d",&n);
    arr = malloc(n * sizeof(int));

    printf("\n Populating the array with random numbers... \n");
    for(i=0;i<n;i++)
        arr[i] = rand() % 100; // Populates the array with random numbers between 0 and 99

    clock_t start = clock(); // Start time

    for(k=0; k<n; k++)
    {
        loc=min(arr,k,n); /* find the smallest element location */
        temp=arr[k];
        arr[k]=arr[loc];
        arr[loc]=temp;
    }
    clock_t end = clock(); // End time

    double time_taken = ((double)end - start) / CLOCKS_PER_SEC * 1000; // Time in milli seconds

    /*printf("\n The Sorted Array is: "); // Uncomment if you want to see the sorted list
    for(i=0;i<n;i++)
        printf(" %d ",arr[i]);*/

    printf("\n Time taken to sort the array = %f ms\n", time_taken);

    return 0;
}
```

Output

```
Run 1:
Enter the number of elements : 500
Populating the array with random numbers...
Time taken to sort the array = 1.000000 ms
```

```
Run 2:
Enter the number of elements : 1000
Populating the array with random numbers...
Time taken to sort the array = 8.000000 ms
```

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

```
#include<stdio.h>

// Function to calculate aⁿ using brute force method
int power_bruteforce(int a, int n) {
    int i, result = 1;
    for(i = 0; i < n; i++) {
        result *= a;
    }
    return result;
}

// Function to calculate aⁿ using divide and conquer method
int power_divide_conquer(int a, int n) {
    if (n == 0)
        return 1;
    else if (n % 2 == 0)
        return power_divide_conquer(a * a, n / 2);
    else
        return a * power_divide_conquer(a * a, n / 2);
}

int main() {
    int a, n;
    printf("Enter the value of a and n: ");
    scanf("%d %d", &a, &n);

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

    printf("Result using brute force: %d\n", result_brute);
    printf("Result using divide and conquer: %d\n", result_divide_conquer);

    return 0;
}
```

Output

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

Explanation

- The program begins by defining two separate functions to calculate a^n: one using a brute-force method, and one using a divide-and-conquer approach.
- In the brute force method (power_bruteforce), the function simply multiplies the base a by itself n times.

- In the divide and conquer method (power_divide_conquer), the function splits the problem into smaller subproblems. If n is even, it calculates $(a^2)^{(n/2)}$. If n is odd, it calculates a * $(a^2)^{(n/2)}$. This takes advantage of the property of exponents that says $(a^b)^c = a^{(b*c)}$.
- In main(), the program prompts the user to enter two integers, a and n. It then calculates a^n using both methods and prints the results.
- **Note:** The divide-and-conquer approach will run more efficiently for large values of n, while the brute force method can lead to slow computations for large n. Additionally, the divide-and-conquer method uses recursion, which may cause a stack overflow if n is very large.

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.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

// Function to partition the array
int partition(int A[], int low, int high) {
    int pivot, j, temp, i;
      pivot = low;
      i = low;
      j = high;
      while (i< j)
      {
            while(i<high && A[i]<=A[pivot])
               i++;
            while (A[j]>A[pivot])
               j--;
            if (i< j)
            {
               temp = A[i];
               A[i] = A[j];
               A[j] = temp;
            }
      }
      temp = A[pivot];
      A[pivot] = A[j];
      A[j] = temp;
      return j;
}
// Quick sort function
void quickSort(int A[], int low, int high) {
    if (low < high) {
        int j = partition(A, low, high);
        quickSort(A, low, j - 1);
        quickSort(A, j + 1, high);
    }
}
```

```
int main()
{
    int i, n, *A;
    srand(time(0)); // Seed the random number generator

    printf("Enter the number of elements : ");
    scanf("%d", &n);
    A = malloc(n * sizeof(int));

    printf("\n Populating the array with random numbers... \n");
    for(i = 0; i < n; i++)
        A[i] = rand() % 1000; // Populates the array with random numbers between 0 and 99

    clock_t start = clock(); // Start time

    quickSort(A, 0, n - 1); // Perform quick sort

    clock_t end = clock(); // End time

    double time_taken = ((double)end - start) / CLOCKS_PER_SEC * 1000; // Time in milli seconds

    /* printf("\n The Sorted Array is: "); // Uncomment if you want to see the sorted list
    for(i = 0; i < n; i++)
        printf(" %d ", A[i]); */

    printf("\n Time taken to sort the array = %f milli seconds\n", time_taken);

    return 0;
}
```

Output

```
Run 1:
Enter the number of elements : 1000
Populating the array with random numbers...
Time taken to sort the array = 0.000000 milli
seconds

Run 2:
Enter the number of elements : 5000
Populating the array with random numbers...
Time taken to sort the array = 0.000000 milli
seconds

Run 3:
Enter the number of elements : 10000
Populating the array with random numbers...
Time taken to sort the array = 7.000000 milli
seconds

Run 4:
Enter the number of elements : 15000
Populating the array with random numbers...
Time taken to sort the array = 10.000000 milli
seconds

Run 5:
Enter the number of elements : 20000
Populating the array with random numbers...
Time taken to sort the array = 15.000000 milli
seconds
```

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.

```
#include<stdio.h>

// Function to calculate factorial for brute force method
int factorial(int n) {
    int i, fact = 1;
    for(i = 2; i <= n; i++)
        fact *= i;
    return fact;
}

// Brute force method to find binomial coefficient
int binomialCoeff_bruteForce(int n, int k) {
    return factorial(n) / (factorial(k) * factorial(n - k));
}

// Dynamic programming method to find binomial coefficient
int binomialCoeff_DP(int n, int k) {
    int C[n+1][k+1];
    int i, j;
    // Calculate value of binomial coefficients
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= ((i < k) ? i : k); j++) {
            // Base Cases
            if (j == 0 || 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];
}

int main()
{
    int n, k;
    printf("Enter the values of n and k: ");
    scanf("%d %d", &n, &k);

    int result_bruteForce = binomialCoeff_bruteForce(n, k);
    int result_DP = binomialCoeff_DP(n, k);
    printf("Binomial Coefficient (Brute Force): %d\n", result_bruteForce);
    printf("Binomial Coefficient (Dynamic Programming): %d\n", result_DP);
    return 0;
}
```

Output

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

Explanation

- The program calculates the binomial coefficient, often represented as C(n, k), using both the brute force method and dynamic programming. The binomial coefficient is the number of ways to choose k elements from a set of n elements without considering the order, often encountered in combinatorial mathematics.
- The program begins by defining two functions, binomialCoeff_bruteForce and binomialCoeff_DP, for the brute force method and dynamic programming method respectively.
- The brute force method implemented in binomialCoeff_bruteForce calculates C(n, k) directly from the formula n! / (k!(n-k)!), where ! denotes factorial. It uses a helper function, factorial, to compute factorials of integers.
- The dynamic programming method, implemented in binomialCoeff_DP, uses a bottom-up approach to fill a 2D array. The entry at the ith row and jth column of this array stores the binomial coefficient C(i, j). For each pair (i, j), it uses the recursive formula C(i, j) = C(i-1, j-1) + C(i-1, j) when j is not 0 or i, and assigns C(i, j) = 1 otherwise.
- The main function reads integers n and k from user input, calculates the binomial coefficient using both methods, and prints both results.

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

```
#include<stdio.h>
#include<stdlib.h>
#define INF 99999

// Print the solution matrix
void printSolution(int V, int **D) {
    printf ("The following matrix shows the shortest distances"
            " between every pair of vertices \n");
    int i, j;
    for (i = 0; i < V; i++) {
        for (j = 0; j < V; j++) {
            if (D[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", D[i][j]);
        }
        printf("\n");
    }
}
```

```
// Implementing Floyd Warshall Algorithm
void floyd (int V, int **C) {
    int i, j, k;

    int **D = (int **)malloc(V * sizeof(int *));
    for(i = 0; i < V; i++)
        D[i] = (int *)malloc(V * sizeof(int));

    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            D[i][j] = C[i][j];

    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (D[i][j] > (D[i][k] + D[k][j]))
                    D[i][j] = D[i][k] + D[k][j];
            }
        }
    }
    printSolution(V, D);
}

int main()
{
    int i,j, V;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    // allocate memory for the cost matrix
    int **C = (int **)malloc(V * sizeof(int *));
    for( i = 0; i < V; i++)
        C[i] = (int *)malloc(V * sizeof(int));

    printf("Enter the cost matrix:\n");
    printf("[Enter 99999 for Infinity] \n");
    printf("[Enter 0 for cost(i,i)] \n");
    for( i = 0; i < V; i++)
        for(j = 0; j < V; j++)
            scanf("%d", &C[i][j]);

    floyd(V, C);
    return 0;
}
```

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:
[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
```

Explanation

- This program is an implementation of Floyd's algorithm which is used to find the shortest distances between every pair of vertices in a graph. The graph is represented by a cost matrix where the cost of going from vertex i to vertex j is given by C[i][j]. If there is no direct edge between vertices i and j, then C[i][j] should be input as INF (represented here as 99999). The cost to go from a vertex to itself is always 0, so C[i][i] should be input as 0 for all i.
- The main function starts by asking the user to enter the number of vertices, V, in the graph. It then allocates memory for the cost matrix C using malloc and asks the user to enter the costs for each pair of vertices. These costs are input directly into the cost matrix. Once all the costs have been entered, the main function calls floyd, passing the number of vertices and the cost matrix.
- The floyd function begins by allocating memory for a new matrix D which will be used to hold the shortest distances between each pair of vertices. It then copies the costs from C into D. The function then applies Floyd's algorithm, updating D[i][j] for all pairs of vertices (i, j) by checking if the current value of D[i][j] is greater than the sum of D[i][k] and D[k][j] for each vertex k. If it is, then D[i][j] is updated with the value of D[i][k] + D[k][j].
- Once Floyd's algorithm has been applied and all the shortest distances have been found, the function print Solution is called to print out the resulting distances. This function prints out INF if the distance between a pair of vertices is INF and the actual distance otherwise.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

double bruteForce(int* coef, int n, double x) {
    double sum = 0.0;
    int i;
    for (i = 0; i <= n; i++) {
        sum += coef[i] * pow(x, i);
    }
    return sum;
}
```

Explanation

- This program computes the topological ordering of vertices in a directed acyclic graph (DAG). The program then prompts the user to enter the number of vertices and the adjacency matrix. The adjacency cost matrix is a square matrix where the entry in the ith row and jth column is 1 if there is an edge from vertex i to vertex j, and 0 otherwise. This matrix is used to represent the graph.
- Next it calculates the in-degree of each vertex by summing the entries in each column of the adjacency matrix. Each column in the adjacency matrix corresponds to a vertex in the graph, and the number of 1's in the column is the in-degree of the vertex.
- Finally, it enters a loop that continues until all vertices have been included in the topological sort (count < n). Inside the loop, the program checks each vertex in the graph. If a vertex has in-degree 0 (meaning it has no incoming edges) and it has not been included in the topological sort yet (flag[k] == 0), it is added to the topological sort and printed. The program then goes through each edge from the chosen vertex (every i such that c[k][i] == 1) and decreases the in-degree of the vertex at the other end of the edge. This represents the removal of the chosen vertex and all edges coming from it.

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

```
#include<stdio.h>

void warshalls(int c[][10], int n) {
    int i, j, k;
    for (k = 0; k < n; k++) {
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (c[i][j] || (c[i][k] && c[k][j]))
                    c[i][j] = 1;
            }
        }
    }
    printf("The transitive closure of the graph is:\n");
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++)
            printf("%d ", c[i][j]);
        printf("\n");
    }
}
int main() {
    int c[10][10], n, i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the adjacency cost matrix:\n");
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            scanf("%d", &c[i][j]);
    warshalls(c, n);
    return 0;
}
```

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
```

Explanation

- This program works by implementing Warshall's algorithm to compute the transitive closure of a directed graph.
- The program prompts the user to enter the number of vertices and the adjacency cost matrix. The function warshalls implements Warshall's algorithm. For each vertex k, it checks every pair of vertices (i, j) and updates c[i][j] to 1 if there's a direct edge from i to j or if there's a path from i to j through k. After all updates, the matrix c will represent the transitive closure of the graph, and it is printed out.

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.**

```
#include <stdio.h>

int w[10], d, n, count, x[10], i;

void sum_of_subsets(int s, int k, int r)
{
    x[k] = 1;
    if (s + w[k] == d)
    {
        printf("\nSubset %d = ", ++count);
        for (i = 0; i <= k; i++)
        {
            if (x[i])
                printf("%d ", w[i]);
        }
    }
    else if (s + w[k] + w[k + 1] <= d)
    {
        sum_of_subsets(s + w[k], k + 1, r - w[k]);
    }
```