EXPERIMENT 1:

AIM: WAP to implement linear search and calculate its step counts.

SOFTWARE USED: VS CODE

PSEUDO CODE:

```
FUNCTION linear(N, A, e)
    FOR i = 0 to N-1
        IF A[i] == e
            RETURN i + 1
        END IF
    END FOR

    RETURN NOT_FOUND
END FUNCTION
```

SOURCE CODE:

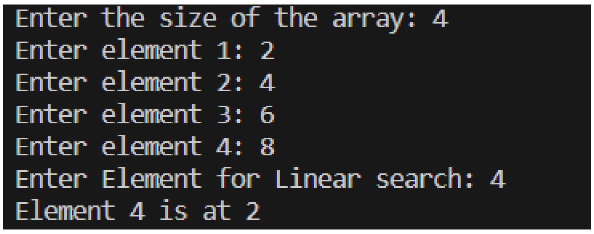
```
#include <stdio.h>
#include <stdlib.h>
int* create(int N)
{
   int * A;
   A = (int *)malloc(100*sizeof(int));
   int i;
   for(i=0;i<N;i++)
   {
      printf("Enter element %d: ",i+1); // 1 step
      scanf("%d",&A[i]); // 1 step
   }
   return (A);
}
int linear(int N,int A[],int e)
{
   for(int i=0;i<N;i++) // 1 step
   {
      if(A[i]==e) // 1 step
      {
         return(i+1); // 1 step
      }
   }
}
int main(){
   int N;
   int * A;
   printf("Enter the size of the array: "); // 1 step
   scanf("%d",&N);
   A = create(N);
```

```
  int e;
  printf("Enter Element for Linear search: "); // 1 step
  scanf("%d",&e);
  printf("Element %d is at %d",e,linear(N,A,e));
  return(0);
}
```

COMPLEXITY:

1+2n +1+2n+1 = 4n + 3 => O(n)

OUTPUT:



```
Enter the size of the array: 4
Enter element 1: 2
Enter element 2: 4
Enter element 3: 6
Enter element 4: 8
Enter Element for Linear search: 4
Element 4 is at 2
```

EXPERIMENT 2:

AIM: WAP to implement binary search and calculate its step counts.

SOFTWARE USED: VS CODE

PSEUDO CODE:

```
FUNCTION binary(N, A, e)
        l = 0
        h = N - 1
        WHILE h >= l
        mid = (h + l) / 2
             IF A[mid] == e
             RETURN mid + 1
             ELSE IF A[mid] > e
                  h = mid - 1
             ELSE IF A[mid] < e
             l = mid + 1
              END IF
        END WHILE

        RETURN NOT_FOUND
END FUNCTION
```

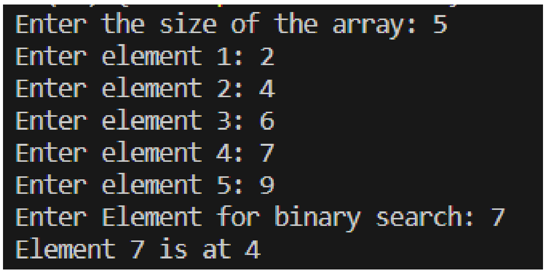
SOURCE CODE:

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

int* create(int N)
{
  int * A;
  A = (int *)malloc(100*sizeof(int));
  int i;
  for(i=0;i<N;i++)
  {
     printf("Enter element %d: ",i+1);// 1 Step
     scanf("%d",&A[i]);// 1 Step
  }
  return (A);
}
int binary(int N, int A[],int e)
{
  int l=0;
  int h=N-1;
  while(h-l>0)
  {
     if(A[(h+l)/2]==e) // 1 Step
     {
        return((h+l)/2+1); // 1 Step
     }
     else if (A[(h+l)/2]>e) // 1 Step
     {
        h=(h+l)/2; // 1 Step
     }
     else if (A[(h+l)/2]<e) // 1 Step
     {
        l=(h+l)/2; // 1 Step
     }
  }
  return(0);
}

int main()
{
  int N;
  int * A;
  printf("Enter the size of the array: "); // 1 Step
  scanf("%d",&N);
  A = create(N); // 1 Step

  int e;

  printf("Enter Element for binary search: "); // 1 Step
  scanf("%d",&e);

  printf("Element %d is at %d\n",e,binary(N,A,e)); // 1 Step
  return(0);
}
```

COMPLEXITY:

1+2n+1+2n+1=4n+3

OUTPUT:



```
Enter the size of the array: 5
Enter element 1: 2
Enter element 2: 4
Enter element 3: 6
Enter element 4: 7
Enter element 5: 9
Enter Element for binary search: 7
Element 7 is at 4
```

EXPERIMENT 2:

AIM: WAP to implement binary search using recursion and calculate its step counts.

SOFTWARE USED: VS CODE

PSEUDO CODE:

```
FUNCTION binary_recursive(l, N, A, e)
    IF N >= l
    mid = (N + l) / 2

    IF A[mid] == e
        RETURN mid + 1
    ELSE IF A[mid] > e
        RETURN binary_recursive(l, mid - 1, A, e)
    ELSE IF A[mid] < e
        RETURN binary_recursive(mid + 1, N, A, e)
    END IF
    END IF

    RETURN NOT_FOUND
END FUNCTION
```

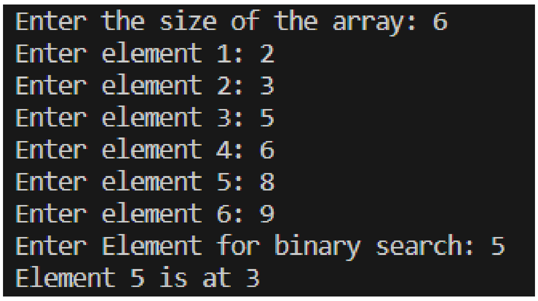
SOURCE CODE:

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

int* create(int N)
{
  int * A;
  A = (int *)malloc(100*sizeof(int));
  int i;
  for(i=0;i<N;i++)
  {
     printf("Enter element %d: ",i+1);// 1 Step
     scanf("%d",&A[i]);// 1 Step
  }
  return (A);
}
int binary_recursive(int l, int N, int A[],int e)
{
  if(A[(N+l)/2]==e)// 1 Step
  {
     return((N+l)/2+1);// 1 Step
  }
  else if (A[(N+l)/2]>e)// 1 Step
  {
    return(binary_recursive(l,(N+l)/2,A,e));// 1 Step
  }
  else if (A[(N+l)/2]<e)// 1 Step
  {
    return(binary_recursive((N+l)/2,N,A,e));// 1 Step
  }
}

int main()
{
  int N;
  int * A;
  printf("Enter the size of the array: ");// 1 Step
  scanf("%d",&N);// 1 Step
  A = create(N);// 1 Step

  int e;

  printf("Enter Element for binary search: ");// 1 Step
  scanf("%d",&e);// 1 Step

  printf("Element %d is at %d\n",e,binary_recursive(0,N,A,e));// 1 Step

  return(0);
}
```

COMPLEXITY:

1+1+2n+1+2n = 4n+3

OUTPUT:



```
Enter the size of the array: 6
Enter element 1: 2
Enter element 2: 3
Enter element 3: 5
Enter element 4: 6
Enter element 5: 8
Enter element 6: 9
Enter Element for binary search: 5
Element 5 is at 3
```

EXPERIMENT 3:

AIM: WAP to implement mergesort and calculate its step counts.

SOFTWARE USED: VS CODE

PSEUDO CODE:

```
mergeSort(arr, s, e):
    if s < e:
        mid = (s + e) / 2
        mergeSort(arr, s, mid)
        mergeSort(arr, mid + 1, e)
        merge(arr, s, mid, e)

merge(arr, s, mid, e):
    n1 = mid - s + 1
    n2 = e - mid
    left[n1], right[n2]

    for i = 0 to n1 - 1:
        left[i] = arr[s + i]
    for j = 0 to n2 - 1:
        right[j] = arr[mid + 1 + j]

    i = 0
    j = 0
    k = s

    while i < n1 and j < n2:
        if left[i] <= right[j]:
            arr[k] = left[i]
            i++
        else:
            arr[k] = right[j]
            j++
        k++

    while i < n1:
        arr[k] = left[i]
        i++
        k++

    while j < n2:
        arr[k] = right[j]
        j++
        k++
```

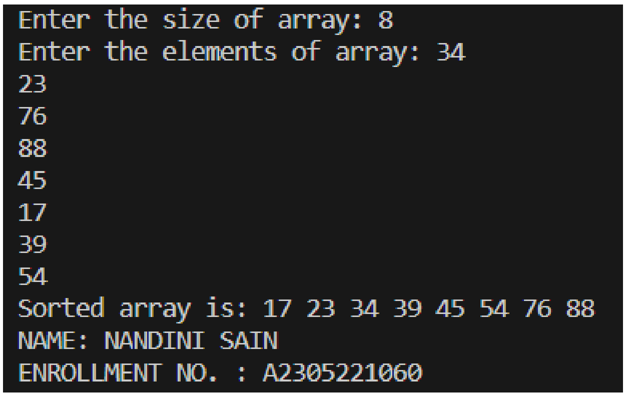
SOURCE CODE:

```
#include <iostream>

using namespace std;

void merge(int arr[], int s, int e)

{

    int mid = (s + e) / 2;

    int i = s;

    int j = mid + 1;

    int k = s;

    int temp[100];

    while (i <= mid && j <= e)

    {

        if (arr[i] < arr[j])

        {

            temp[k++] = arr[i++];

        }

        else

        {

            temp[k++] = arr[j++];

        }

    }

    while (i <= mid)

    {

        temp[k++] = arr[i++];

    }

    while (j <= e)

    {

        temp[k++] = arr[j++];

    }

    for (int i = s; i <= e; i++)

    {

        arr[i] = temp[i];
    }
}
void mergesort(int arr[], int s, int e)
{
    if (s < e)
    {
        int mid = (s + e) / 2;
        mergesort(arr, s, mid);
        mergesort(arr, mid + 1, e);
        merge(arr, s, e);
    }
}
int main()
{
    int n;
    cout << "Enter the size of array: ";
    cin >> n;
    int arr[n];
    cout << "Enter the elements of array: ";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    mergesort(arr, 0, n - 1);
    cout << "Sorted array is: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

COMPLEXITY:

9 + 7n + 3nlogn = O(nlogn)

OUTPUT:



```
Enter the size of array: 8
Enter the elements of array: 34
23
76
88
45
17
39
54
Sorted array is: 17 23 34 39 45 54 76 88
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 4:

AIM: WAP to implement quicksort and calculate its step counts.

SOFTWARE USED: VS CODE

PSEUDO CODE:

```
quicksort(arr, s, e):
    if s < e:
        p = partition(arr, s, e)
        quicksort(arr, s, p - 1)
        quicksort(arr, p + 1, e)
partition(arr, s, e):
    pivot = arr[e]
    i = s - 1
    for j = s to e - 1:
        if arr[j] < pivot:
            i++
            swap(arr[i], arr[j])
    swap(arr[i + 1], arr[e])
    return i + 1
```

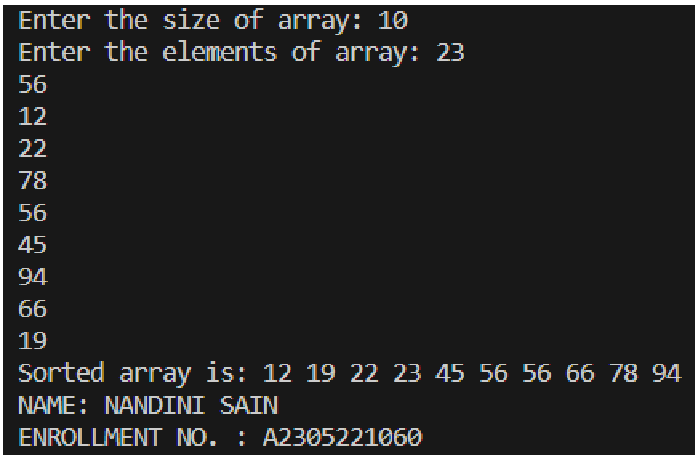
SOURCE CODE:

```
#include <iostream>
using namespace std;
int partition(int arr[], int s, int e) //n
{
    int pivot = arr[e]; //1
    int i = s - 1; //1
    for (int j = s; j < e; j++) //n
    {
        if (arr[j] < pivot) //1
        {
            i++; //1
            swap(arr[i], arr[j]);  //1
        }
    }
    swap(arr[i + 1], arr[e]); //1
    return i + 1; //1
}
void quicksort(int arr[], int s, int e) //logn
{
    if (s < e)  //1
    {
        int p = partition(arr, s, e); //n
        quicksort(arr, s, p - 1);
        quicksort(arr, p + 1, e);
    }
}
int main()
{
    int n; //1
    cout << "Enter the size of array: "; //1
    cin >> n; //1
    int arr[n]; //1
    cout << "Enter the elements of array: "; //1
    for (int i = 0; i < n; i++) //n
    {
        cin >> arr[i]; //n
    }
    quicksort(arr, 0, n - 1); //logn
    cout << "Sorted array is: "; //1
    for (int i = 0; i < n; i++) //n
    {
        cout << arr[i] << " "; //n
    }
    return 0; //1
}
```

COMPLEXITY:

9n + 2logn + 20=O(nlogn)

OUTPUT:



```
Enter the size of array: 10
Enter the elements of array: 23
56
12
22
78
56
45
94
66
19
Sorted array is: 12 19 22 23 45 56 56 66 78 94
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 5:

AIM: WAP to implement insertion sort and calculate its step counts.

SOFTWARE USED: VS CODE

PSEUDO CODE:

```
insertionSort(arr):
    n = length of arr
    for i = 1 to n - 1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j--
        arr[j + 1] = key
```

SOURCE CODE:

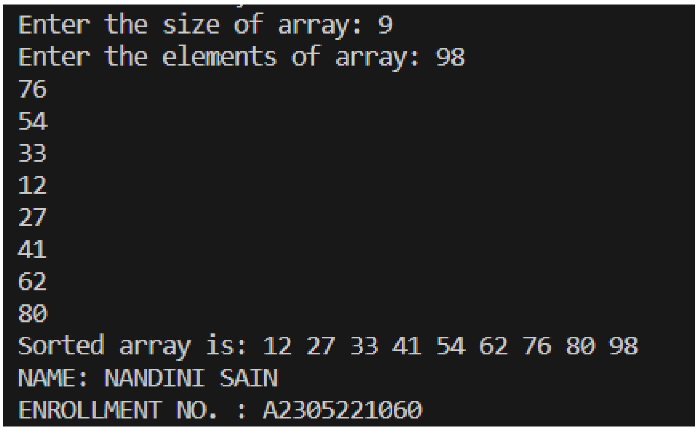
```
#include <iostream>
using namespace std;
void insertion_sort(int arr[], int n)
{
    for (int i = 1; i < n; i++) //n
    {
        int current = arr[i]; //1
        int j = i - 1; //1
        while (arr[j] > current && j >= 0) //n
        {
            arr[j + 1] = arr[j]; //1
            j--;
        }
        arr[j + 1] = current; //1
    }
}
int main()
```

```
{
   int n; //1
   cout << "Enter the size of array: "; //1
   cin >> n; //1
   int arr[n]; //1
   cout << "Enter the elements of array: "; //1
   for (int i = 0; i < n; i++) //n
   {
      cin >> arr[i]; //n
   }
   insertion_sort(arr, n); //n^2
   cout << "Sorted array is: "; //1
   for (int i = 0; i < n; i++) //n
   {
      cout << arr[i] << " "; //n
   }
   printf("\nNAME: NANDINI SAIN");
   printf("\nENROLLMENT NO. : A2305221060");
   return 0; //1
}
```

COMPLEXITY:

9n^2 + 9n + 10= O(n^2)

OUTPUT:



```
Enter the size of array: 9
Enter the elements of array: 98
76
54
33
12
27
41
62
80
Sorted array is: 12 27 33 41 54 62 76 80 98
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 6:

AIM: WAP to implement selection sort and calculate its step counts.
SOFTWARE USED: VS CODE
PSEUDO CODE:

```
selectionSort(arr):
    n = length of arr
    for i = 0 to n - 2:
        minIndex = i
        for j = i + 1 to n - 1:
            if arr[j] < arr[minIndex]:
                minIndex = j
        swap(arr[i], arr[minIndex])
```

SOURCE CODE:

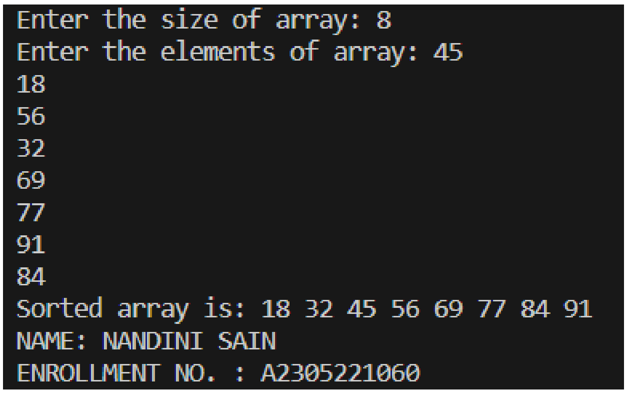
```
#include <iostream>
using namespace std;
void selection_sort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min = i; //1
        for (int j = i + 1; j < n; j++) //n
        {
            if (arr[j] < arr[min]) //1
            {
                min = j; //1
            }
        }
        swap(arr[i], arr[min]); //1
    }
}
int main()
```

```
{
    int n; //1
    cout << "Enter the size of array: "; //1
    cin >> n; //1
    int arr[n]; //1
    cout << "Enter the elements of array: "; //1
    for (int i = 0; i < n; i++) //n
    {
        cin >> arr[i]; //n
    }
    selection_sort(arr, n); //n^2
    cout << "Sorted array is: "; //1
    for (int i = 0; i < n; i++) //n
    {
        cout << arr[i] << " "; //n
    }
    printf("\nNAME: NANDINI SAIN");
    printf("\nENROLLMENT NO. : A2305221060");
    return 0; //1
}
```

COMPLEXITY:

6n^2 + 6n + 7 = O(n^2)

OUTPUT:



```
Enter the size of array: 8
Enter the elements of array: 45
18
56
32
69
77
91
84
Sorted array is: 18 32 45 56 69 77 84 91
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 7:

AIM: WAP to implement bubble sort and calculate its step counts.
SOFTWARE USED: VS CODE
PSEUDO CODE:

```
bubbleSort(arr):
    n = length of arr
    for i = 0 to n - 1:
        for j = 0 to n - i - 2:
            if arr[j] > arr[j + 1]:
                swap(arr[j], arr[j + 1])
```

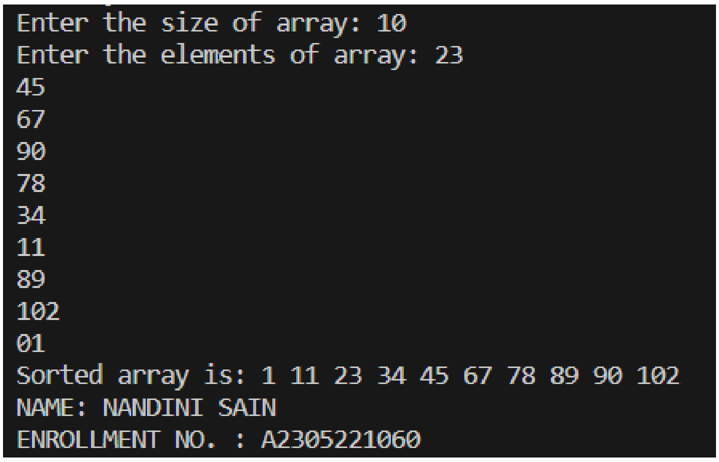
SOURCE CODE:

```
#include <iostream>
using namespace std;

void bubble_sort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++) //n
    {
        for (int j = 0; j < n - i - 1; j++) //n
        {
            if (arr[j] > arr[j + 1]) //1
            {
                swap(arr[j], arr[j + 1]); //1
            }
        }
    }
}

int main()
{
    int n; //1
    cout << "Enter the size of array: "; //1
    cin >> n; //1
    int arr[n]; //1
    cout << "Enter the elements of array: "; //1
    for (int i = 0; i < n; i++) //n
    {
        cin >> arr[i]; //n
    }
    bubble_sort(arr, n); //n^2
    cout << "Sorted array is: "; //1
    for (int i = 0; i < n; i++) //n
    {
        cout << arr[i] << " "; //n
    }
    printf("\nNAME: NANDINI SAIN");
    printf("\nENROLLMENT NO. : A2305221060");
    return 0; //1
}
```

COMPLEXITY

6n^2 + 6n + 7 = O(n^2)

OUTPUT:



```
Enter the size of array: 10
Enter the elements of array: 23
45
67
90
78
34
11
89
102
01
Sorted array is: 1 11 23 34 45 67 78 89 90 102
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 8:

AIM: WAP to implement Fractional Knapsack algorithm.

SOFTWARE USED: VS CODE

PSEUDO CODE:

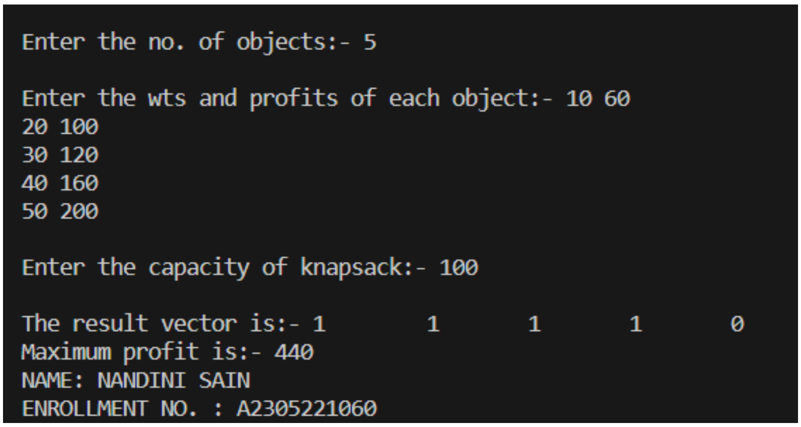
```
FractionalKnapsack(items, capacity):
   Sort items by decreasing profit-to-weight ratio
   totalProfit = 0
   knapsack = []
   for each item in items:
      if capacity == 0:
         break
      fraction = min(item.weight, capacity)
      totalProfit = totalProfit + fraction * item.profit
      capacity = capacity - fraction
      knapsack.append({item: fraction})
   return totalProfit, knapsack
```

SOURCE CODE:

```
#include <iostream>
using namespace std;
void knapsack(int n, float weight[], float profit[], float capacity)
{
   float x[20], tp = 0;
   int i, j, u;
   u = capacity;
   for (i = 0; i < n; i++)
      x[i] = 0.0;
   for (i = 0; i < n; i++)
   {
      if (weight[i] > u)
         break;
      else
```

```
        {
            x[i] = 1.0;
            tp = tp + profit[i];
            u = u - weight[i];
        }
    }
    if (i < n)
        x[i] = u / weight[i];
    tp = tp + (x[i] * profit[i]);
    cout << "\nThe result vector is:- ";
    for (i = 0; i < n; i++)
        cout << x[i] << "\t";
    cout << "\nMaximum profit is:- " << tp;
}
int main()
{
    float weight[20], profit[20], capacity;
    int num, i, j;
    float ratio[20], temp;
    cout << "\nEnter the no. of objects:- ";
    cin >> num;
    cout << "\nEnter the wts and profits of each object:- ";
    for (i = 0; i < num; i++)
    {
        cin >> weight[i] >> profit[i];
    }
    cout << "\nEnter the capacity of knapsack:- ";
    cin >> capacity;
    for (i = 0; i < num; i++)
    {
        ratio[i] = profit[i] / weight[i];
    }
    for (i = 0; i < num; i++)
    {
        for (j = i + 1; j < num; j++)
        {
            if (ratio[i] < ratio[j])
            {
                temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;
                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;
                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }
    knapsack(num, weight, profit, capacity);
    printf("\nNAME: NANDINI SAIN");
    printf("\nENROLLMENT NO. : A2305221060");
    return 0;
}
```

OUTPUT



```
Enter the no. of objects:- 5

Enter the wts and profits of each object:- 10 60
20 100
30 120
40 160
50 200

Enter the capacity of knapsack:- 100

The result vector is:- 1        1       1       1       0
Maximum profit is:- 440
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 9:

AIM: WAP to implement Kruskal’s algorithm.
SOFTWARE USED: VS CODE
PSEUDO CODE:

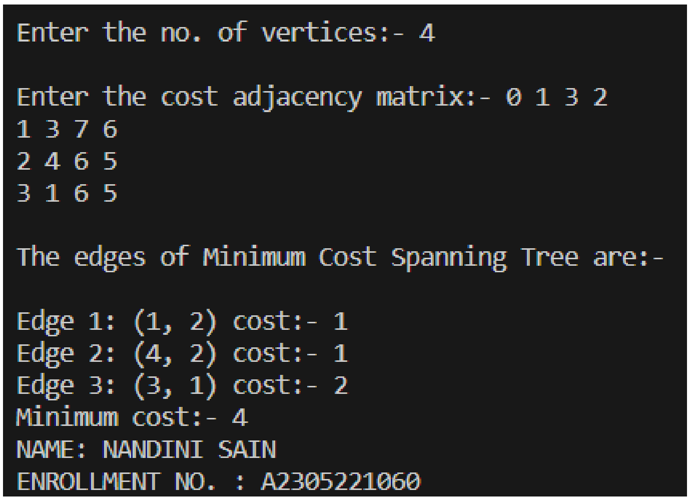
```
KruskalMST(graph):
   Sort all edges in increasing order of their weight
   Initialize an empty set for the Minimum Spanning Tree (MST)

   for each vertex v in the graph:
      Make a set containing only vertex v

   Initialize an empty list to store selected edges

   for each edge (u, v, w) in the sorted edges:
      if the sets containing u and v are not the same:
         Add edge (u, v, w) to the MST
         Union the sets containing u and v

   return the MST
```

SOURCE CODE:

```
#include<iostream>
using namespace std;
int main()
{
   int i, j, k, a, b, u, v, n, ne = 1;
   int min, mincost = 0, cost[9][9], parent[9];
   cout << "\nEnter the no. of vertices:- ";
   cin >> n;
   cout << "\nEnter the cost adjacency matrix:- ";
   for (i = 1; i <= n; i++)
```

```
{
        for (j = 1; j <= n; j++)
            cin >> cost[i][j];
        parent[i] = 0;
    }
    cout << "\nThe edges of Minimum Cost Spanning Tree are:- " << endl;
    while (ne < n)
    {
        for (i = 1, min = 999; i <= n; i++)
        {
            for (j = 1; j <= n; j++)
            {
                if (cost[i][j] < min)
                {
                    min = cost[i][j];
                    a = u = i;
                    b = v = j;
                }
            }
        }
        while (parent[u])
            u = parent[u];

        while (parent[v])
            v = parent[v];

        if (u != v)
        {
            cout << "\nEdge " << ne++ << ": (" << a << ", " << b << ") cost:- " << min;
            mincost += min;
            parent[v] = u;
        }
        cost[a][b] = cost[b][a] = 999;
    }
    cout << "\nMinimum cost:- " << mincost << endl;
    cout<< "NAME: NANDINI SAIN"<<endl;
    cout<< "ENROLLMENT NO. : A2305221060"<<endl;
    return 0;
}
```

OUTPUT:



```
Enter the no. of vertices:- 4

Enter the cost adjacency matrix:- 0 1 3 2
1 3 7 6
2 4 6 5
3 1 6 5

The edges of Minimum Cost Spanning Tree are:-

Edge 1: (1, 2) cost:- 1
Edge 2: (4, 2) cost:- 1
Edge 3: (3, 1) cost:- 2
Minimum cost:- 4
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 10:

AIM: WAP to implement Prim’s algorithm.
SOFTWARE USED: VS CODE
PSEUDO CODE:

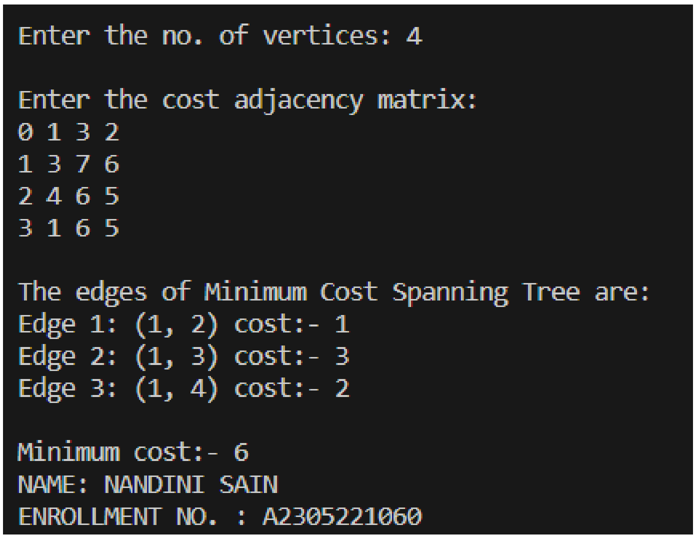
```
PrimMST(graph):
   Initialize an empty set to store the Minimum Spanning Tree (MST)
   Initialize a set containing the starting vertex


   while the MST does not include all vertices:
      Find the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST
      Add the edge's destination vertex to the MST
      Add the edge to the MST


   return the MST
```

SOURCE CODE:

```
#include <iostream>
#include <climits>
using namespace std;
int main() {
   int i, j, k, a, b, u, v, n;
   int mincost = 0, cost[9][9], parent[9], key[9];
   bool mstSet[9];
   cout << "\nEnter the no. of vertices: ";
   cin >> n;
   cout << "\nEnter the cost adjacency matrix:\n";
   for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
         cin >> cost[i][j];
      }
      parent[i] = -1;
      key[i] = INT_MAX;
```

```
        mstSet[i] = false;
    }
    key[1] = 0;
    for (i = 1; i <= n; i++) {
        int minKey = INT_MAX, u;
        for (v = 1; v <= n; v++) {
            if (!mstSet[v] && key[v] < minKey) {
                minKey = key[v];
                u = v;
            }
        }
        mstSet[u] = true;
        mincost += key[u];
        for (v = 1; v <= n; v++) {
            if (cost[u][v] && !mstSet[v] && cost[u][v] < key[v]) {
                parent[v] = u;
                key[v] = cost[u][v];
            }
        }
    }
    cout << "\nThe edges of Minimum Cost Spanning Tree are:\n";
    for (i = 2; i <= n; i++) {
        cout << "Edge " << i - 1 << ": (" << parent[i] << ", " << i << ") cost:- " << 
cost[parent[i]][i] << endl;
    }
    cout << "\nMinimum cost:- " << mincost << endl;
    cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;
    return 0;
}
```

OUTPUT:



```
Enter the no. of vertices: 4

Enter the cost adjacency matrix:
0 1 3 2
1 3 7 6
2 4 6 5
3 1 6 5

The edges of Minimum Cost Spanning Tree are:
Edge 1: (1, 2) cost:- 1
Edge 2: (1, 3) cost:- 3
Edge 3: (1, 4) cost:- 2

Minimum cost:- 6
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 11:

AIM: WAP to implement Dijkrsta’s algorithm.
SOFTWARE USED: VS CODE
PSEUDO CODE:

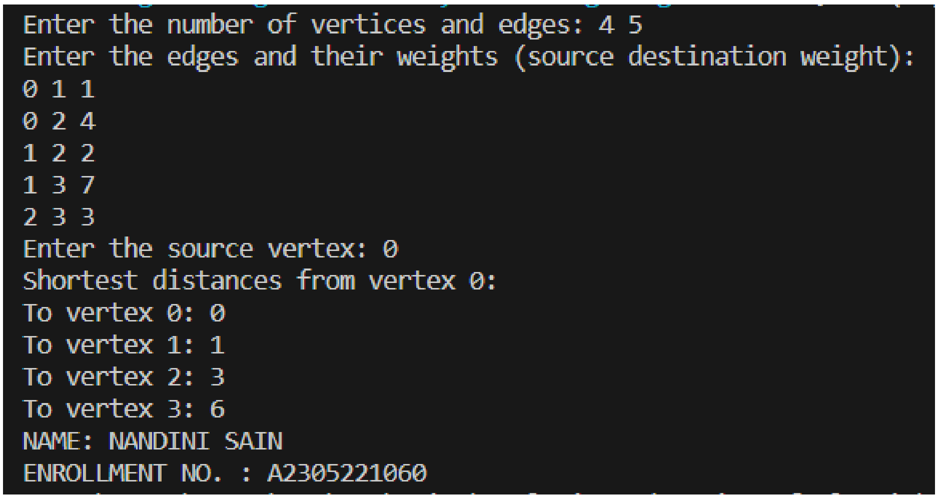
```
Dijkstra's Algorithm:

Input: Graph G, source vertex s

Output: Shortest distances from s to all vertices in G


1. Create a set S to keep track of vertices whose shortest distances are finalized. Initialize it as an empty set.

2. Create an array dist[] of size |V| (number of vertices) and initialize it with infinity (∞). Set dist[s] = 0 because the distance from s to itself is zero.

3. Create a priority queue (min-heap) Q to store vertices with their tentative distances. Initialize it with all vertices and their corresponding dist[] values.

4. While Q is not empty:

  a. Extract the vertex u with the minimum dist[u] from Q.

  b. Add u to set S to mark it as finalized.

  c. For each neighbor v of u:

    i. If v is not in set S:

      - Calculate the tentative distance new_dist from s to v via u (dist[u] + weight(u, v)).

      - If new_dist is less than dist[v], update dist[v] with new_dist.

      - Update the priority of v in Q with new_dist.

5. Return the array dist[], which contains the shortest distances from s to all vertices in G.
```

SOURCE CODE:

```
#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

const int INF = INT_MAX;

struct Edge {

   int to;

   int weight;
```

```
};
void dijkstra(vector<vector<Edge>>& graph, int start, vector<int>& dist) {
    int V = graph.size();
    dist.assign(V, INF);
    dist[start] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, start});
    while (!pq.empty()) {
        int u = pq.top().second;
        int u_dist = pq.top().first;
        pq.pop();
        if (u_dist > dist[u]) continue;
        for (const Edge& edge : graph[u]) {
            int v = edge.to;
            int weight = edge.weight;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
}
int main() {
    int V, E;
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;
    vector<vector<Edge>> graph(V);
    cout << "Enter the edges and their weights (source destination weight):" << endl;
    for (int i = 0; i < E; i++) {
        int src, dest, weight;
        cin >> src >> dest >> weight;
        graph[src].push_back({dest, weight});
        graph[dest].push_back({src, weight});  // Assuming an undirected graph.
    }
    int start;
    cout << "Enter the source vertex: ";
    cin >> start;
    vector<int> dist;
    dijkstra(graph, start, dist);
    cout << "Shortest distances from vertex " << start << ":" << endl;
    for (int i = 0; i < V; i++) {
        cout << "To vertex " << i << ": " << dist[i] << endl;
    }
    cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;
    return 0;
}
```

OUTPUT:



```
Enter the number of vertices and edges: 4 5
Enter the edges and their weights (source destination weight):
0 1 1
0 2 4
1 2 2
1 3 7
2 3 3
Enter the source vertex: 0
Shortest distances from vertex 0:
To vertex 0: 0
To vertex 1: 1
To vertex 2: 3
To vertex 3: 6
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 12:

AIM: WAP to implement Strassen’s Multiplication.
SOFTWARE USED: VS CODE
PSEUDO CODE:

```
function strassenMatrixMultiply(A, B):
    if size(A) is 1:
        # Base case: Single element multiplication
        return A * B

    # Split matrices A and B into four equal-sized submatrices
    A11, A12, A21, A22 = splitMatrix(A)
    B11, B12, B21, B22 = splitMatrix(B)

    # Recursively calculate seven products (P1 to P7)
    P1 = strassenMatrixMultiply(A11, subtractMatrix(B12, B22))
    P2 = strassenMatrixMultiply(addMatrix(A11, A12), B22)
    P3 = strassenMatrixMultiply(addMatrix(A21, A22), B11)
    P4 = strassenMatrixMultiply(A22, subtractMatrix(B21, B11))
    P5 = strassenMatrixMultiply(addMatrix(A11, A22), addMatrix(B11, B22))
    P6 = strassenMatrixMultiply(subtractMatrix(A12, A22), addMatrix(B21, B22))
    P7 = strassenMatrixMultiply(subtractMatrix(A11, A21), addMatrix(B11, B12))

    # Calculate the resulting submatrices C11, C12, C21, and C22
    C11 = subtractMatrix(addMatrix(addMatrix(P5, P4), P6), P2)
    C12 = addMatrix(P1, P2)
    C21 = addMatrix(P3, P4)
    C22 = subtractMatrix(subtractMatrix(addMatrix(P5, P1), P3), P7)

    # Combine submatrices into the resulting matrix C
    C = combineMatrices(C11, C12, C21, C22)

    return C
```

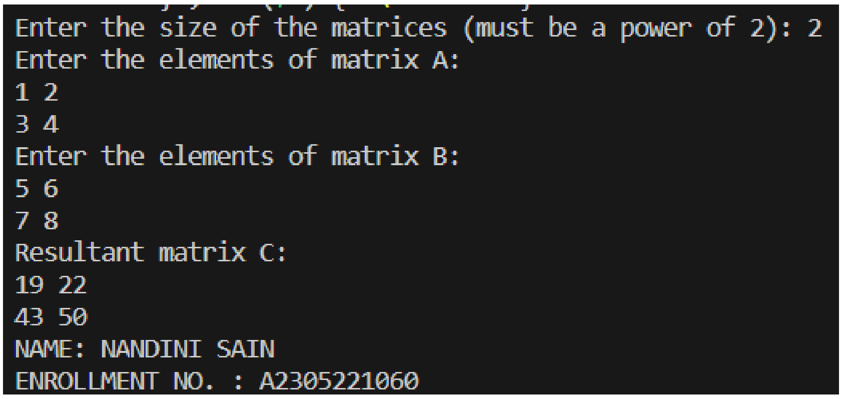
SOURCE CODE:

```
#include <iostream>

using namespace std;


// Function to add two matrices

void addMatrices(int** A, int** B, int** C, int n) {

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            C[i][j] = A[i][j] + B[i][j];
        }
    }
}
// Function to subtract two matrices
void subtractMatrices(int** A, int** B, int** C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
}
// Function to multiply two matrices using Strassen's algorithm
void strassenMultiply(int** A, int** B, int** C, int n) {
    if (n <= 2) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                C[i][j] = 0;
                for (int k = 0; k < n; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        return;
    }
    // Divide matrices into four submatrices
    int mid = n / 2;
    int** A11 = new int*[mid];
    int** A12 = new int*[mid];
    int** A21 = new int*[mid];
    int** A22 = new int*[mid];
    int** B11 = new int*[mid];
    int** B12 = new int*[mid];
```

```
int** B21 = new int*[mid];
int** B22 = new int*[mid];

for (int i = 0; i < mid; i++) {
   A11[i] = new int[mid];
   A12[i] = new int[mid];
   A21[i] = new int[mid];
   A22[i] = new int[mid];
   B11[i] = new int[mid];
   B12[i] = new int[mid];
   B21[i] = new int[mid];
   B22[i] = new int[mid];
}
int** P1 = new int*[mid];
int** P2 = new int*[mid];
int** P3 = new int*[mid];
int** P4 = new int*[mid];
int** P5 = new int*[mid];
int** P6 = new int*[mid];
int** P7 = new int*[mid];

for (int i = 0; i < mid; i++) {
   P1[i] = new int[mid];
   P2[i] = new int[mid];
   P3[i] = new int[mid];
   P4[i] = new int[mid];
   P5[i] = new int[mid];
   P6[i] = new int[mid];
   P7[i] = new int[mid];
}
```

```
    // Calculate the result submatrices
    int** C11 = new int*[mid];
    int** C12 = new int*[mid];
    int** C21 = new int*[mid];
    int** C22 = new int*[mid];

    for (int i = 0; i < mid; i++) {
        C11[i] = new int[mid];
        C12[i] = new int[mid];
        C21[i] = new int[mid];
        C22[i] = new int[mid];
    }

int main() {
    int n;
    cout << "Enter the size of the matrices (must be a power of 2): ";
    cin >> n;
    int** A = new int*[n];
    int** B = new int*[n];
    int** C = new int*[n];

    for (int i = 0; i < n; i++) {
        A[i] = new int[n];
        B[i] = new int[n];
        C[i] = new int[n];
    }
    cout << "Enter the elements of matrix A:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> A[i][j];
        }
    }
    cout << "Enter the elements of matrix B:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cin >> B[i][j];
        }
    }
    if ((n & (n - 1)) != 0) {
        cout << "Matrix size must be a power of 2." << endl;
        return 1;
    }
    strassenMultiply(A, B, C, n);
    cout << "Resultant matrix C:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << C[i][j] << " ";
        }
        cout << endl;
    }
    cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;
    return 0;
}
```

OUTPUT:



```
Enter the size of the matrices (must be a power of 2): 2
Enter the elements of matrix A:
1 2
3 4
Enter the elements of matrix B:
5 6
7 8
Resultant matrix C:
19 22
43 50
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 13:

AIM: WAP to implement Lowest Common Subsequence.
SOFTWARE USED: VS CODE
PSEUDO CODE:

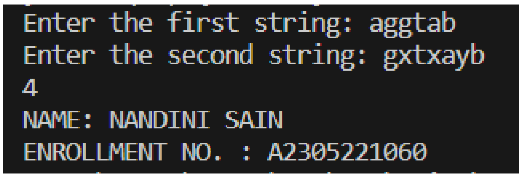
```
function LCSLength(s1, s2):
    m = length of s1
    n = length of s2
    // Initialize a 2D array dp[m+1][n+1] to store LCS lengths.
    dp = new int[m+1][n+1]
    for i from 0 to m:
        for j from 0 to n:
            if i == 0 or j == 0:
                // Base case: LCS with an empty string is 0.
                dp[i][j] = 0
            else if s1[i-1] == s2[j-1]:
                // If the characters match, extend the LCS.
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                // Characters don't match, take the maximum of the previous LCS values.
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```

SOURCE CODE:

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int lcs(string s1, string s2) {
    int m = s1.length();
    int n = s2.length();
    int** dp = new int*[m + 1];
    for (int i = 0; i <= m; i++) {
        dp[i] = new int[n + 1];
        for (int j = 0; j <= n; j++) {
            dp[i][j] = 0;
        }
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            // If the last characters match, add 1 to the result
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            }
            // Otherwise, take the maximum of the two possibilities
            else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    int result = dp[m][n];

    for (int i = 0; i <= m; i++) {
        delete[] dp[i];
    }
    delete[] dp;

    return result;
}

    int main() {
```

```
    string s1, s2;
    cout << "Enter the first string: ";
    cin >> s1;
    cout << "Enter the second string: ";
    cin >> s2;

    int length = lcs(s1, s2);

    cout << lcs(s1, s2) << endl;
    cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;

    return 0;
}
```

OUTPUT:



```
Enter the first string: aggtab
Enter the second string: gxtxayb
4
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 14:

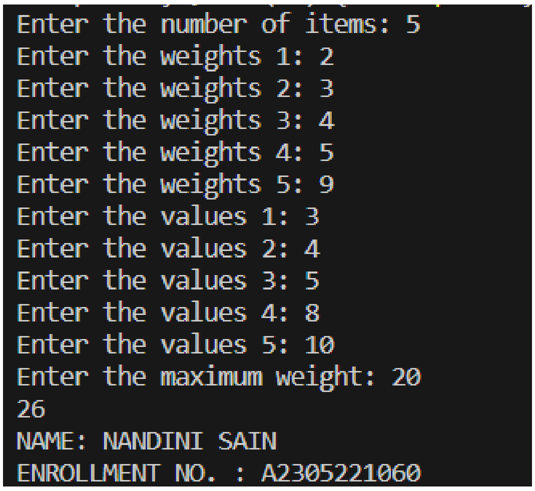
AIM: WAP to implement Knapsack using dynamic programming.
SOFTWARE USED: VS CODE
PSEUDO CODE:

```
KnapsackDP(weights[], values[], capacity, n)
  Create a 2D array dp of size (n + 1) x (capacity + 1) and initialize it with zeros.
  for i from 0 to n do
    for w from 0 to capacity do
      if i is 0 or w is 0 then
        dp[i][w] = 0
      else if weights[i - 1] <= w then
        dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
      else
        dp[i][w] = dp[i - 1][w]
  Initialize variables i to n and w to capacity.
  Create an empty array selectedItems.
  while i > 0 and w > 0 do
    if dp[i][w] is not equal to dp[i - 1][w] then
      Add weights[i - 1] to selectedItems
      Subtract weights[i - 1] from w
    Decrement i by 1
  return dp[n][capacity] (maximum value) and selectedItems (items selected in the knapsack)
```

SOURCE CODE:

```
#include <iostream>
#include <algorithm>
using namespace std;

int knapsack(int* weights, int* values, int n, int maxWeight) {
    int dp[n + 1][maxWeight + 1];
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= maxWeight; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            } else {
                int inc = 0;
                int exc = 0;

                if (j >= weights[i - 1]) {
                    inc = values[i - 1] + dp[i - 1][j - weights[i - 1]];
                }
                exc = dp[i - 1][j];
                dp[i][j] = max(inc, exc);
            }
        }
    }
    return dp[n][maxWeight];
}

int main() {
    int n;
    cout << "Enter the number of items: ";
    cin >> n;

    int* weights = new int[n];
    int* values = new int[n];

    for (int i = 0; i < n; i++) {
        cout << "Enter the weights " << i + 1 << ": ";
        cin >> weights[i];
    }
    for (int i = 0; i < n; i++) {
        cout << "Enter the values " << i + 1 << ": ";
        cin >> values[i];
    }

    int maxWeight;
    cout << "Enter the maximum weight: ";
    cin >> maxWeight;

    cout << knapsack(weights, values, n, maxWeight) << endl;
    delete[] weights;
    delete[] values;

    cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;
    return 0;
}
```

OUTPUT:



```
Enter the number of items: 5
Enter the weights 1: 2
Enter the weights 2: 3
Enter the weights 3: 4
Enter the weights 4: 5
Enter the weights 5: 9
Enter the values 1: 3
Enter the values 2: 4
Enter the values 3: 5
Enter the values 4: 8
Enter the values 5: 10
Enter the maximum weight: 20
26
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 15:

AIM: WAP to implement Breadth First Search.
SOFTWARE USED: VS CODE
PSEUDO CODE:

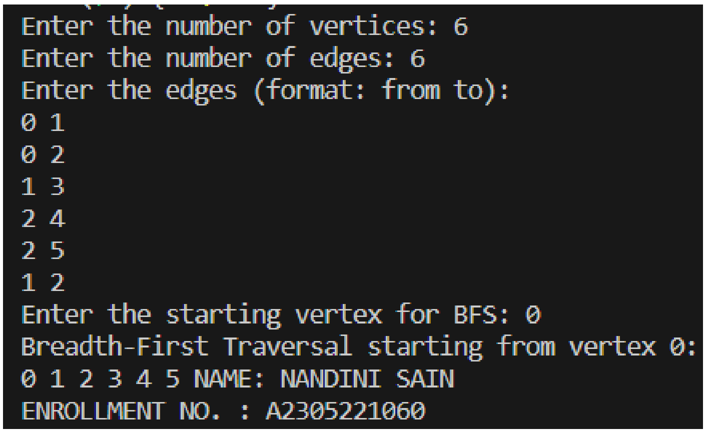
```
function bfs(start_node, num_nodes):
 Initialize an empty queue.
 Create a boolean array 'visited' of size 'num_nodes' and initialize it to all False.
 Mark the 'start_node' as visited and enqueue it in the queue.
 while the queue is not empty:
 Dequeue the front node 'curr_node' from the queue.
 Print 'curr_node' to indicate it has been visited.
 for each neighboring node 'i' from 0 to 'num_nodes - 1':
 if there is an edge from 'curr_node' to 'i' (adj_matrix[curr_node][i] == 1) and 'i' has not been visited (visited[i] is False):
 Mark 'i' as visited (visited[i] = True).
 Enqueue 'i' in the queue.
function main():
 Input the number of nodes 'num_nodes'.
 Create an adjacency matrix 'adj_matrix' of size 'MAX_NODES x MAX_NODES'.
 Input the adjacency matrix values representing connections between nodes.
 Input the starting node for BFS 'start_node'.
 Call the 'bfs' function with 'start_node' and 'num_nodes' to perform the BFS traversal.
 Output the order of visited nodes.
main()
```

SOURCE CODE:

```
#include <iostream>
#include <queue>
using namespace std;

// Define the maximum number of vertices
const int MAX_VERTICES = 100;
int graph[MAX_VERTICES][MAX_VERTICES]; // Adjacency matrix
```

```
bool visited[MAX_VERTICES];            // To keep track of visited nodes
// Function to add an edge to the graph
void addEdge(int from, int to) {
   graph[from][to] = 1;
   graph[to][from] = 1; // For an undirected graph
}
// BFS function
void bfs(int start, int vertices) {
   queue<int> q;
   visited[start] = true;
   q.push(start);
   while (!q.empty()) {
      int current = q.front();
      cout << current << " ";
      q.pop();
      for (int i = 0; i < vertices; i++) {
         if (graph[current][i] && !visited[i]) {
            visited[i] = true;
            q.push(i);
         }
      }
   }
}
int main() {
   int vertices, edges;
   cout << "Enter the number of vertices: ";
   cin >> vertices;
   cout << "Enter the number of edges: ";
   cin >> edges;
   // Initialize the graph and visited array
   for (int i = 0; i < vertices; i++) {
      for (int j = 0; j < vertices; j++) {
            graph[i][j] = 0;
        }
        visited[i] = false;
    }
    cout << "Enter the edges (format: from to):" << endl;
    for (int i = 0; i < edges; i++) {
        int from, to;
        cin >> from >> to;
        addEdge(from, to);
    }
    int startVertex;
    cout << "Enter the starting vertex for BFS: ";
    cin >> startVertex;
    cout << "Breadth-First Traversal starting from vertex " << startVertex << ": ";
    bfs(startVertex, vertices);
    cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;
    return 0;
}
```

OUTPUT:



```
Enter the number of vertices: 6
Enter the number of edges: 6
Enter the edges (format: from to):
0 1
0 2
1 3
2 4
2 5
1 2
Enter the starting vertex for BFS: 0
Breadth-First Traversal starting from vertex 0:
0 1 2 3 4 5 NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 16:

AIM: WAP to implement Depth First Search.

SOFTWARE USED: VS CODE

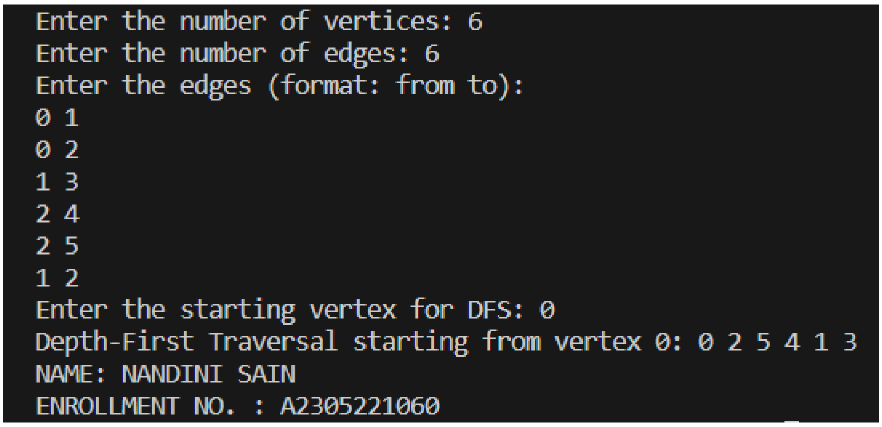
PSEUDO CODE:

```
function DFS(node):
 Mark the current node as visited.
 Print the current node.
 For each neighboring node i:
 If i is connected to the current node and i has not been visited:
 Recursively call DFS(i) to visit node i.
Main Function:
 Input the number of nodes in the graph (num_nodes).
 Input the adjacency matrix (graph) representing the connections between nodes.
 Specify the starting node for DFS (start_node).
 Perform DFS traversal starting from start_node using the DFS function.
 Output the order of visited nodes.
```

SOURCE CODE:

```
#include <iostream>
#include <stack>
using namespace std;

// Define the maximum number of vertices
const int MAX_VERTICES = 100;
int graph[MAX_VERTICES][MAX_VERTICES]; // Adjacency matrix
bool visited[MAX_VERTICES];           // To keep track of visited nodes
// Function to add an edge to the graph
void addEdge(int from, int to) {
   graph[from][to] = 1;
   graph[to][from] = 1; // For an undirected graph
}
// DFS function
void dfs(int start, int vertices) {
   stack<int> s;
   visited[start] = true;
   s.push(start);
   while (!s.empty()) {
       int current = s.top();
       cout << current << " ";
       s.pop();
       for (int i = 0; i < vertices; i++) {
           if (graph[current][i] && !visited[i]) {
               visited[i] = true;
               s.push(i);
           }
       }
   }
}
int main() {
   int vertices, edges;
   cout << "Enter the number of vertices: ";
   cin >> vertices;
   cout << "Enter the number of edges: ";
   cin >> edges;
   // Initialize the graph and visited array
   for (int i = 0; i < vertices; i++) {
       for (int j = 0; j < vertices; j++) {
           graph[i][j] = 0;
       }
       visited[i] = false;
   }
   cout << "Enter the edges (format: from to):" << endl;
   for (int i = 0; i < edges; i++) {
       int from, to;
       cin >> from >> to;
       addEdge(from, to);
   }
   int startVertex;
   cout << "Enter the starting vertex for DFS: ";
   cin >> startVertex;
   cout << "Depth-First Traversal starting from vertex " << startVertex << ": ";
   dfs(startVertex, vertices);
   cout << endl;
   cout << "NAME: NANDINI SAIN" << endl;
   cout << "ENROLLMENT NO. : A2305221060" << endl;
   return 0;
}
```

OUTPUT:



```
Enter the number of vertices: 6
Enter the number of edges: 6
Enter the edges (format: from to):
0 1
0 2
1 3
2 4
2 5
1 2
Enter the starting vertex for DFS: 0
Depth-First Traversal starting from vertex 0: 0 2 5 4 1 3
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```

EXPERIMENT 17:

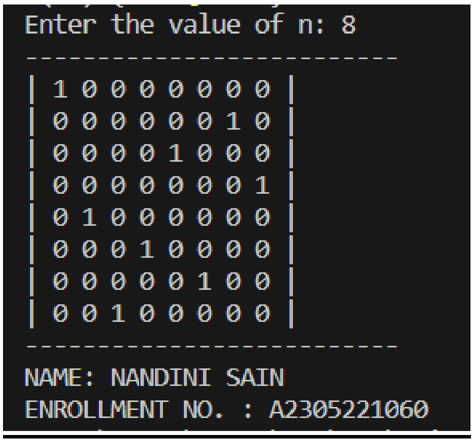
AIM: WAP to implement N Queen.
SOFTWARE USED: VS CODE
PSEUDO CODE:

```
function solveNQueens(N):
    Initialize an empty chessboard[N][N]
    if placeQueens(chessboard, 0, N) returns true:
        Print chessboard as the solution
    else:
        Print "No solution exists"
function placeQueens(chessboard, col, N):
    if col >= N:
        # All queens are placed successfully
        return true
    for each row from 0 to N-1:
        if isSafe(chessboard, row, col, N):
            chessboard[row][col] = "Q"  # Place a queen
            if placeQueens(chessboard, col + 1, N):
                return true
            chessboard[row][col] = "."  # Backtrack
    return false
function isSafe(chessboard, row, col, N):
    # Check if it's safe to place a queen at chessboard[row][col]
    # Check the left side of this row
    for i from 0 to col - 1:
        if chessboard[row][i] == "Q":
            return false
    # Check upper-left diagonal
    for i from row, j from col to 0:
        if chessboard[i][j] == "Q":
            return false
    # Check lower-left diagonal
    for i from row, j from col to 0, i < N:
        if chessboard[i][j] == "Q":
            return false
    return true
```

SOURCE CODE:

```
#include <iostream>
using namespace std;

#define N 8

int board[N][N];

void printSolution(int n) {
    cout << "-------------------------" << endl;
    for (int i = 0; i < n; i++) {
        cout << "| ";
        for (int j = 0; j < n; j++) {
            cout << board[i][j] << " ";
        }
        cout << "|" << endl;
    }
    cout << "-------------------------" << endl;
}

bool isSafe(int row, int col, int n) {
    int i, j;

    // Check the column on the left side
    for (i = 0; i < col; i++) {
        if (board[row][i]) return false;
    }
    // Check upper left diagonal
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) return false;
    }

    // Check lower left diagonal
    for (i = row, j = col; j >= 0 && i < n; i++, j--) {
        if (board[i][j]) return false;
    }

    return true;
}

bool solveNQueens(int col, int n) {
    if (col >= n) return true; // All queens are placed successfully

    for (int i = 0; i < n; i++) {
        if (isSafe(i, col, n)) {
            board[i][col] = 1; // Place queen

            if (solveNQueens(col + 1, n)) return true; // Recur to place the rest of the queens

            board[i][col] = 0; // If placing queen doesn't lead to a solution, backtrack
        }
    }

    return false; // If queen can't be placed in any row, return false
}

int main() {
    int n;
    cout << "Enter the value of n: ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            board[i][j] = 0; // Initialize board with all zeros
        }
    }

    if (solveNQueens (0, n)) {
        printSolution(n);
    } else {
        cout << "No solution found!" << endl;
    }
     cout << "NAME: NANDINI SAIN" << endl;
    cout << "ENROLLMENT NO. : A2305221060" << endl;
    return 0;
}
```

OUTPUT:



```
Enter the value of n: 8
-------------------------
| 1 0 0 0 0 0 0 0 |
| 0 0 0 0 0 0 1 0 |
| 0 0 0 0 1 0 0 0 |
| 0 0 0 0 0 0 0 1 |
| 0 1 0 0 0 0 0 0 |
| 0 0 0 1 0 0 0 0 |
| 0 0 0 0 0 1 0 0 |
| 0 0 1 0 0 0 0 0 |
-------------------------
NAME: NANDINI SAIN
ENROLLMENT NO. : A2305221060
```