

Practical : 6

Problem Statement : Write a C++ program to show function overloading.

Implementation :

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

// function with float type parameter
float absolute(float var)
{
    if (var < 0.0)
        var = -var;
    return var;
}

// function with int type parameter
int absolute(int var)
{
    if (var < 0)
        var = -var;
    return var;
}

int main()
{
    // call function with int type parameter
    cout << "Absolute value of -25 = " << absolute(-25) << endl;

    // call function with float type parameter
    cout << "Absolute value of 15.5 = " << absolute(15.5f) << endl;
    return 0;
}
```

Result :

```
Absolute value of -25 = 25
Absolute value of 15.5 = 15.5
```

Practical : 7

Problem Statement : Write a C++ program to demonstrate the use of constructor and destructor.

Implementation :

```
#include<bits/stdc++.h>
using namespace std;
class Distance
{
private:
    int feet;
    int inches;

public:
    Distance() {

    }
    Distance(int f, int i)
    {
        feet = f;
        inches = i;
    }
    void get_distance()
    {
        cout << "Distance is feet= " << feet << ", inches= " << inches <<
endl;
    }
    void add(Distance &d1, Distance &d2)
    {
        feet = d1.feet + d2.feet;
        inches = d1.inches + d2.inches;
        feet = feet + (inches / 12);
        inches = inches % 12;
    }
    ~Distance()
    {
        cout << "Distance object destroyed" << endl;
    }
};
int main()
{
    int f1, in1, f2, in2;
    cout << "Enter feet: ";
```

```

    cin >> f1;
    cout << "Enter inches: ";
    cin >> in1;
    cout << "Enter feet: ";
    cin >> f2;
    cout << "Enter inches: ";
    cin >> in2;
    Distance d1(f1, in1);
    Distance d2(f2, in2);
    Distance d3;
    d3.add(d1, d2);
    d3.get_distance();
    return 0;
}

```

Result :

```

PS E:\All NOTES 2nd YEAR\All NOTES 2nd YEAR\DSLab\DSLab\cp folder>
p.cpp -o cp } ; if ($?) { .\cp }
Enter feet: 5
Enter inches: 4
Enter feet: 6
Enter inches: 1
Distance is feet= 11, inches= 5
Distance object destroyed
Distance object destroyed
Distance object destroyed

```

Practical : 8

Problem Statement : . Write a C++ program to implement virtual function and pure virtual function.

Implementation :

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

class base
{
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base
{
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
    base *p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
    p->fun_1();

    // Late binding (RTP)
```

```

p->fun_2();

// Late binding (RTP)
p->fun_3();

// Late binding (RTP)
p->fun_4();

// Early binding but this function call is
// illegal(produces error) because pointer
// is of base type and function is of
// derived class
// p->fun_4(5);
}

```

Result :

```

p.cpp -o cp }
base-1
derived-2
base-3
base-4

```

Program using Pure Virtual Function :

```

#include <iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
class Derived : public Base
{
public:
    void show() { cout << "In Derived \n"; }
};
int main(void)

```

```
{  
    Base *bp = new Derived();  
    bp->show();  
    return 0;  
}
```

Result :

In Derived

Practical : 9

Problem Statement : Write a C++ program to swap two numbers using friend function

Implementation :

```
#include <iostream>
using namespace std;
class Swap
{
    // Declare the variables of Swap Class
    int temp, a, b;
public:
    // Define the parameterized constructor, for inputs
    Swap(int a, int b)
    {
        this->a = a;
        this->b = b;
    }
    // Declare the friend function to swap, take arguments
    // as call by reference
    friend void swap(Swap &);
};

// Define the swap function outside class scope
void swap(Swap &s1)
{
    // Call by reference is used to passed object copy to
    // the function
    cout << "\nBefore Swapping: " << s1.a << " " << s1.b;

    // Swap operations with Swap Class variables
    s1.temp = s1.a;
    s1.a = s1.b;
    s1.b = s1.temp;
    cout << "\nAfter Swapping: " << s1.a << " " << s1.b;
}

int main()
```

```
{  
    // Declare and Initialize the Swap object  
    Swap s(4, 6);  
    swap(s);  
    return 0;  
}
```

Result :

```
Before Swapping: 4 6  
After Swapping: 6 4
```


Practical : 10

Problem Statement : Write a C++ program to add two complex numbers using the concept called operator overloading.

Implementation :

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

class Complex
{
public:
    float real;
    float img;

    Complex()
    {
        this->real = 0.0;
        this->img = 0.0;
    }

    Complex(float real, float img)
    {
        this->real = real;
        this->img = img;
    }

    // overloading + operator
    Complex operator+(const Complex &obj)
    {
        Complex temp;
        temp.img = this->img + obj.img;
        temp.real = this->real + obj.real;
        return temp;
    }

    void display()
    {
        cout << this->real << " + " << this->img << "i" << endl;
    }
};

int main()
{
```

```

Complex a, b, c;

cout << "Enter real and complex coefficient of the first complex
number: " << endl;
cin >> a.real;
cin >> a.img;

cout << "Enter real and complex coefficient of the second complex
number: " << endl;
cin >> b.real;
cin >> b.img;

cout << "Addition Result: ";
c = a + b;
c.display();

return 0;
}

```

Result :

```

PS E:\All NOTES 2nd YEAR\All NOTES 2nd YEAR\DSLAb\DSLAb\cp folder
p.cpp -o cp } ; if ($?) { .\cp }
Enter real and complex coefficient of the first complex number:
24 10
Enter real and complex coefficient of the second complex number:
21 4
Addition Result: 45 + 14i

```

EXPERIMENT : 8

Problem Statement: Write a program to sort an array using HeapSort.

Theory : Heap sort is an efficient sorting algorithm with $O(n \log n)$ worst case time complexity. It is an in place sorting algorithm.

Algorithm:

HEAPSORT(*A*)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
```

BUILD-MAX-HEAP(*A*)

```
1  A.heap-size = A.length
2  for i =  $\lfloor A.length/2 \rfloor$  downto 1
3      MAX-HEAPIFY(A, i)
```

MAX-HEAPIFY(*A*, *i*)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if  $l \leq A.heap-size$  and  $A[l] > A[i]$ 
4      largest = l
5  else largest = i
6  if  $r \leq A.heap-size$  and  $A[r] > A[largest]$ 
7      largest = r
8  if largest  $\neq i$ 
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)
```

PARENT(*i*)

```
1  return  $\lfloor i/2 \rfloor$ 
```

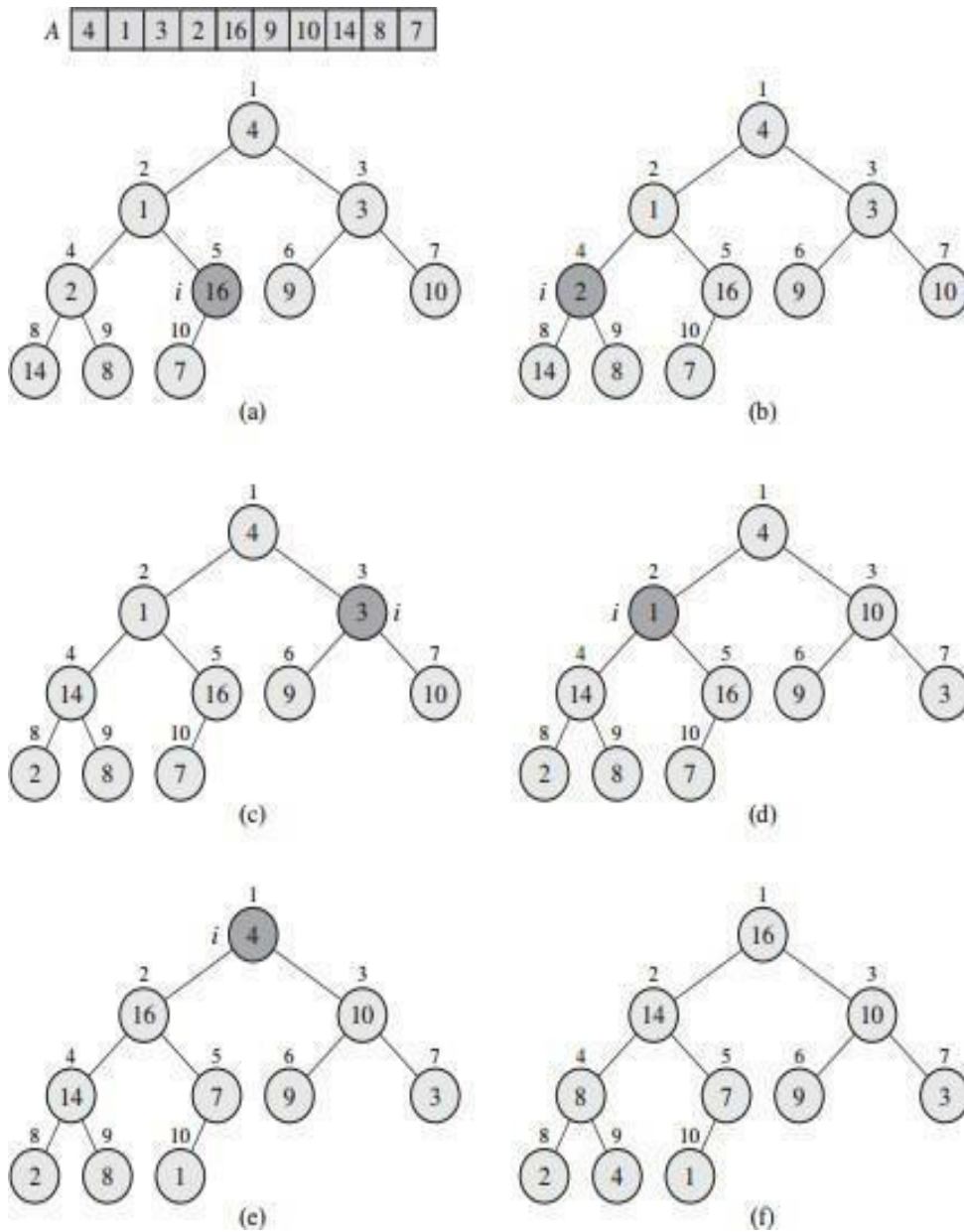
LEFT(*i*)

```
1  return  $2i$ 
```

RIGHT(*i*)

```
1  return  $2i + 1$ 
```

Example showing operations of BUILD MAX HEAP:



Implementation :

```
#include <iostream>
using namespace std;
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
void heapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
int main()
{
    int n;
    cout << "Enter size of Array-:";
    cin >> n;
    int array[n];
    cout << "Enter " << n << " numbers: " << endl;
    for (int i = 0; i < n; i++)
    {
        cin >> array[i];
    }
}
```

```

    }
    cout << "\nWithout Sort-:" << endl;
    for (int j = 0; j < n; j++)
    {
        cout << array[j] << " ";
    }
    heapSort(array, n);

    cout << "\nSorted array is \n";
    for (int j = 0; j < n; j++)
    {
        cout << array[j] << " ";
    }
}

```

Output:

```

Enter size of Array-:5
Enter 5 numbers:
7
0
1
8
3

Without Sort-:
7 0 1 8 3
Sorted array is
0 1 3 7 8
Process returned 0 (0x0)   execution time : 6.311 s
Press any key to continue.

```

EXPERIMENT : 9

Problem Statement: Write a program to solve the fractional knapsack problem.

Theory: Fractional knapsack is another good example to explain greedy technique. The knapsack problem can be used to find the optimal solution to fill the knapsack with maximum profit. Likewise, we can apply the same technique to obtain the optimal solution for eating a meal. So that, we can get maximum calories by using the optimal solution.

Inputs: Number of objects n , Profit $P(1:n)$ and weight $W(1:n)$ for n objects, Knapsack size (M)

Output: Solution vector $X(1:n)$ showing which object is selected and in what fraction.

Algorithm:

Sort the objects in descending order on the basis of profit(p_i)/weight(w_i) from $[1 \dots n]$. then call GREEDY_KNAPSACK.

```
procedure GREEDY_KNAPSACK( $P, W, M, X, n$ )
  //  $P(1:n)$  and  $W(1:n)$  contain the profits and weights respectively of the  $n$  //
  // objects ordered so that  $P(i)/W(i) \geq P(i+1)/W(i+1)$ .  $M$  is the //
  // knapsack size and  $X(1:n)$  is the solution vector //
  real  $P(1:n), W(1:n), X(1:n), M, cu$ ;
  integer  $i, n$ ;
   $X \leftarrow 0$  // initialize solution to zero //
   $cu \leftarrow M$  //  $cu$  = remaining knapsack capacity //
  for  $i \leftarrow 1$  to  $n$  do
    if  $W(i) > cu$  then exit endif
     $X(i) \leftarrow 1$ 
     $cu \leftarrow cu - W(i)$ 
  repeat
    if  $i \leq n$  then  $X(i) \leftarrow cu/W(i)$  endif
  end GREEDY_KNAPSACK
```

Implementation :

```
#include <iostream>
#include<conio.h>
using namespace std;

void knapsack(int value[], int weight[], float maxi, int n)
{
    float current = 0, rem, currentval = 0;
    int i = 0;
    while ((current <= maxi) && (weight[i] <= maxi - current))
    {
        current = current + weight[i];
        currentval = currentval + value[i];
        i++;
    }
    rem = maxi - current;
    if (current < maxi)
    {
        current = current + rem;
        currentval = currentval + ((rem * value[i]) / weight[i]);
    }

    cout << "\n\nMaximumprofitis:" << currentval;
    cout << "\n Current weight is:" << current;
}

main()
{
    int n;
    cout << "Enter the number of item: ";
    cin >> n;
    int value[n], weight[n], maxi, i;
    cout << "Enter theCapacity ofKnapsack:";
    cin >> maxi;
    cout << "Enterthevaluesofitem-:\n";
    for (i = 0; i < n; i++)
    {
        cin >> value[i];
    }
    cout << "EnterthevaluesofWeight-:\n";
    for (i = 0; i < n; i++)
```



```

{
    cin >> weight[i];
}
cout << "Elements Enter arw-:\n";
for (i = 0; i < n; i++)
{
    cout << "\nValue " << value[i] << " Weight " << weight[i];
}
knapsack(value, weight, maxi, n);
}

```

Output:

```

"C:\Users\Shivank Varshney\Desktop\s.exe"
Enter the number of item: 4
Enter the Capacity of Knapsack: 60
Enter the values of item-:
50
60
70
70
Enter the values of Weight-:
10
20
30
40
Elements Enter arw-:

Value 50 Weight 10
Value 60 Weight 20
Value 70 Weight 30
Value 70 Weight 40

Maximum profit is: 180
Current weight is: 60
Process returned 0 (0x0)   execution time : 17.465 s
Press any key to continue.

```

EXPERIMENT : 10

Problem Statement: Write a Program to implement travelling Salesperson problem.

Theory : Travelling Salesperson problem is good example of dynamic programming. It is focused on optimization. TSP is a mathematical problem. It is most easily expressed as a graph describing the locations of a set of nodes.

Inputs: Input is a graph $G(V, E)$ with weight w for each edge in E .

Outputs: Returns the Minimum Path from starting node by traversing all the nodes.

Implementation :

```
#include<stdio.h>
int ary[10][10],completed[10],n,cost=0;
void takeInput()
{
    int i,j;
    printf("Enter the number of villages: ");
    scanf("%d",&n);
    printf("\nEnter the Cost Matrix\n");
    for(i=0;i < n;i++)
    {
        printf("\nEnter Elements of Row: %d\n",i+1);
        for( j=0;j < n;j++)
            scanf("%d",&ary[i][j]);
        completed[i]=0;
    }
    printf("\n\nThe cost list is:");
    for( i=0;i < n;i++)
    {
        printf("\n");
        for(j=0;j < n;j++)
            printf("\t%d",ary[i][j]);
    }
}
void mincost(int city)
{
    int i, ncity;
    completed[city]=1;
    printf("%d--->",city+1);
```

```

        ncity=least(city);
        if(ncity==999)
        {
            ncity=0;
            printf("%d",ncity+1);
            cost+=ary[city][ncity];
            return;
        }
        mincost(ncity);
    }
}

int least(int c)
{
    int i,nc=999;
    int min=999,kmin;
    for(i=0;i < n;i++)
    {
        if((ary[c][i]!=0)&&(completed[i]==0))
        if(ary[c][i]+ary[i][c] < min)
        {
            min=ary[i][0]+ary[c][i];
            kmin=ary[c][i];
            nc=i;
        }
    }
    if(min!=999)
    cost+=kmin;
    return nc;
}

int main()
{
    takeInput();
    printf("\n\nThe Path is:\n");
    mincost(0);

    printf("\n\nMinimum cost is %d\n ",cost);
    return 0;
}

```

Output:

```
Enter Elements of Row: 1
0 4 1 3
```

```
Enter Elements of Row: 2
4 0 2 1
```

```
Enter Elements of Row: 3
1 2 0 5
```

```
Enter Elements of Row: 4
3 1 5 0
```

```
The cost list is:
      0      4      1      3
      4      0      2      1
      1      2      0      5
      3      1      5      0
```

```
The Path is:
1--->3--->2--->4--->1
```

```
Minimum cost is 7
```

EXPERIMENT : 11

Problem Statement: WAP to find the minimum cost spanning tree of a given graph $G(V,E)$ and weight of each edge using Kruskals algorithm.

Theory : Kruskals algorithm is good example of greedy technique. Minimum spanning trees were first studied for ways to lay out electrical networks in a way that minimizes the total cost of the wiring.

Inputs: Input is a graph $G(V,E)$ with weight w for each edge in E .

Outputs: Returns the set of edges in the minimum cost spanning tree and the minimum cost

Algorithm:

MST-KRUSKAL(G,w)

```
1  A=φ
2  for each vertex  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  Sort the edges of  $G.E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u,v) \in G.E$ , taken in nondecreasing order by weight
6      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7          A=A
            U  $\{(u,v)\}$ 
8      UNION( $u,v$ )
9  return A
```

It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u . Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

Implementation :

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,parent[9000];
int find(int i)
{
    while(parent[i])
        i=parent[i];
    return i;
}
int uni(int i,int j)
{
    if(i!=j)
    {
        parent[j]=i;
        return 1;
    }
    return 0;
}
void main()
{
    printf("\nEnter the no. of vertices:");
    scanf("%d",&n);
    int cost[n][n];
    printf("\nEnter the cost adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=999;
        }
    }
    printf("\nThe edges of Minimum Cost Spanning Tree are\n");
    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;
    }

}

}

u=find(u);
v=find(v);
if(uni(u,v))
{
    printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
    mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
}

```

Output:

```

C:\Users\Shivank Varshney\Documents\kruskal.exe
Enter the no. of vertices:4
Enter the cost adjacency matrix:
1
2
3
4
5
6
/
3
9
10
11
12
13
14
15
16
The edges of Minimum Cost Spanning Tree are
1 edge (1,2) =2
2 edge (1,3) =3
3 edge (1,4) =4

Minimum cost = 9

Process returned 19 (0x13)   execution time : 20.834 s
Press any key to continue.

```

EXPERIMENT : 12

Problem Statement: Write a program to implement N Queen problem using Backtracking.

Theory : The N Queen is the problem of placing N chess queens on an $N \times N$ chessboard so that no two queens attack each other

Inputs: A $n * n$ Chessboard and n queens..

Outputs: A binary matrix which has 1s for the blocks where **queens** are placed.

Algorithm:

1. Place the queens column wise, start from the left most column
2. If all queens are placed.
 1. return true and print the solution matrix.
3. Else
 1. Try all the rows in the current column.
 2. Check if queen can be placed here safely if yes mark the current cell in solution matrix as 1 and try to solve the rest of the problem recursively.
 3. If placing the queen in above step leads to the solution return true.
 4. If placing the queen in above step does not lead to the solution , BACKTRACK, mark the current cell in solution matrix as 0 and return false.
4. If all the rows are tried and nothing worked, return false and print NO SOLUTION.

Possible Outcomes:

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```


Implementation :

```
#include <iostream>
using namespace std;
#define N 8
void printBoard(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl;
    }
}
bool isValid(int board[N][N], int row, int col)
{
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    for (int i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;
    return true;
}
bool solveNQueen(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if (isValid(board, i, col))
        {
            board[i][col] = 1;
            if (solveNQueen(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
}
```

```
        return false;
    }
int main()
{
    int board[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            board[i][j] = 0;

    if (solveNQueen(board, 0) == false)
    {
        cout << "Solution does not exist";
    }
    printBoard(board);
}
```

Output:

```
PS E:\All NOTES 2nd YEAR\All NOTES 2nd YEAR\DSLab\DSLab\cp folder> cd
p.cpp -o cp } ; if ($?) { .\cp }
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
PS E:\All NOTES 2nd YEAR\All NOTES 2nd YEAR\DSLab\DSLab\cp folder>
```

EXPERIMENT : 13

Problem Statement: Write a Program to implement 0/1 Knapsack problem using Dynamic Programming.

Theory : A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i . What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either 0 or 1 , where other constraints remain the same.

Algorithm:

Let i be the highest-numbered item in an optimal solution S for W dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution S is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define $c[i, w]$ to be the solution for items $1, 2, \dots, i$ and the maximum weight w .

The algorithm takes the following inputs

- The maximum weight W
- The number of items n
- The two sequences $v = \langle v_1, v_2, \dots, v_n \rangle$ and $w = \langle w_1, w_2, \dots, w_n \rangle$

Dynamic-0-1-knapsack (v, w, n, W)

for $w = 0$ to W do

$c[0, w] = 0$

for $i = 1$ to n

do $c[i, 0] = 0$

for $w = 1$ to W

do if $w_i \leq w$ then

if $v_i + c[i-1, w-w_i] > c[i-1, w]$

then $c[i, w] = v_i + c[i-1, w-w_i]$

else

$c[i, w] = c[i-1, w]$ else

$c[i, w] = c[i-1, w]$

The set of items to take can be deduced from the table, starting at $c[n, w]$ and tracing backwards where the optimal values came from.

If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise, item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.

Implementation :

```
#include <iostream>
using namespace std;
int max(int a, int b)
{
    return (a > b) ? a : b;
}
// Returns the maximum profit
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n + 1][W + 1];
    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    cout << "maximum profit is:" << knapSack(W, wt, val, n);
    return 0;
}
```

Output:

```
maximum profit is: 220
```

```
-----
```

```
Process exited after 1.014 seconds with return value 0
```

```
Press any key to continue . . .
```

EXPERIMENT : 14

Problem Statement: From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.

Theory : Dijkstra's algorithm is based on greedy approach. Dijkstra's Algorithm finds the shortest path between a given node (which is called the "source node") and all other nodes in a graph. This algorithm uses the weights of the edges to **find** the path that minimizes the total distance (weight) between the source node and all other nodes. Time Complexity of Dijkstra's Algorithm is $O(V + E \log V)$

Algorithm:

```
DIJKSTRA(G,w,s)
  INITIALIZE-SINGLE-SOURCE(G,S)
  S =  $\Phi$ 
  Q = V[G]
  while Q  $\neq \Phi$ 
    u = EXTRACT-MIN(Q)
    S = S  $\cup$  {u}
    for each vertex v  $\in$  G.adj[u]
      RELAX(u,v,w)
```

Implementation :

```
#include <stdio.h>
#include <conio.h>
#define INFINITY 9999
#define MAX 10
void dijkstra(int G[MAX][MAX], int n, int startnode);
int main()
{
    int G[MAX][MAX], i, j, n, u;
    printf("Enter no. of vertices:");
    scanf("%d", &n);
    printf("\nEnter the adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &G[i][j]);
    printf("\nEnter the starting node:");
    scanf("%d", &u);
    dijkstra(G, n, u);
    return 0;
}
```

```

void dijkstra(int G[MAX][MAX], int n, int startnode)
{
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(G[i][j]==0)
                cost[i][j]=INFINITY;
    else
        cost[i][j] = G[i][j];
    // initialize pred[], distance[] and visited[]
    for(i=0;i<n;i++)
    {
        distance[i] = cost[startnode][i];
        pred[i] = startnode;
        visited[i] = 0;
    }
    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;
    while (count < n - 1)
    {
        mindistance = INFINITY;
        // nextnode gives the node at minimum distance
        for(i=0;i<n;i++)
            if(distance[i]<mindistance&&!visited[i])
            {
                mindistance = distance[i];
                nextnode = i;
            }
        // check if a better path exists through nextnode
        visited[nextnode]=1;
        for (i = 0; i < n; i++)
            if (!visited[i])
                if (mindistance + cost[nextnode][i] < distance[i])
                {
                    distance[i] = mindistance + cost[nextnode][i];
                    pred[i] = nextnode;
                }
        count++;
    }
}

```



```

        // print the path and distance of each node
for(i=0;i<n;i++)
    if (i != startnode)
    {
        printf("\nDistance of node%d=%d", i, distance[i]);
        printf("\nPath=%d", i);
        j = i;
        do
        {
            j = pred[j];
            printf("<-%d", j);
        } while (j != startnode);
    }
}
}

```

Output:

```

Enter the adjacency matrix:
0
10
0
30
100
10
0
50
0
0
0
50
0
20
10
30
0
20
0
60
100
0
10
60
0

Enter the starting node:0

Distance of node1=10
Path=1<-0
Distance of node2=50
Path=2<-3<-0
Distance of node3=30
Path=3<-0
Distance of node4=60
Path=4<-2<-3<-0

```

EXPERIMENT : 15

Problem Statement: WAP to find the minimum cost spanning tree of a given graph $G(V,E)$ and weight of each edge using Prim's algorithm.

Objectives: Prim's algorithm is a good example of greedy technique. Minimum spanning trees were first studied for ways to lay out electrical networks in a way that minimizes the total cost of the wiring.

Inputs: Input is a graph $G(V,E)$ with weight w for each edge in E .

Outputs: Returns the set of edges in the minimum cost spanning tree and the minimum cost

Algorithm:

MST-Prim's(G,w)

- 1 $A = \emptyset$
- 2 **for** each vertex $v \in G.V$
- 3 MAKE-SET(v)
- 4 Sort the edges of $G.E$ into nondecreasing order by weight w
- 5 **for** each edge $(u,v) \in G.E$, taken in nondecreasing order by weight
- 6 **if** FIND-SET(u) \neq FIND-SET(v)
- 7 $A = A \cup \{(u,v)\}$
- 8 UNION(u,v)
- 9 return A

It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u . Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

Implementation :

```
#include <stdio.h>
#include <conio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10] = {0}, min, mincost = 0, cost[10][10];
int main()
{

    printf("\nEnter the number of nodes:");
    scanf("%d", &n);
    printf("\nEnter the adjacency matrix:\n");
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= n; j++)

        {

            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = 999;
        }
    }

    visited[1] = 1;
    printf("\n");
    while (ne < n)
    {

        for (i = 1, min = 999; i <= n; i++)
            for (j = 1; j <= n; j++)
                if (cost[i][j] < min)
                    if (visited[i] != 0)

                    {

                        min = cost[i][j];
                        a = u = i;
                        b = v = j;
                    }

        if (visited[u] == 0 || visited[v] == 0)

        {

            printf("\n Edge %d:(%d %d) cost:%d", ne++, a, b, min);
            mincost += min;
        }
    }
}
```

```

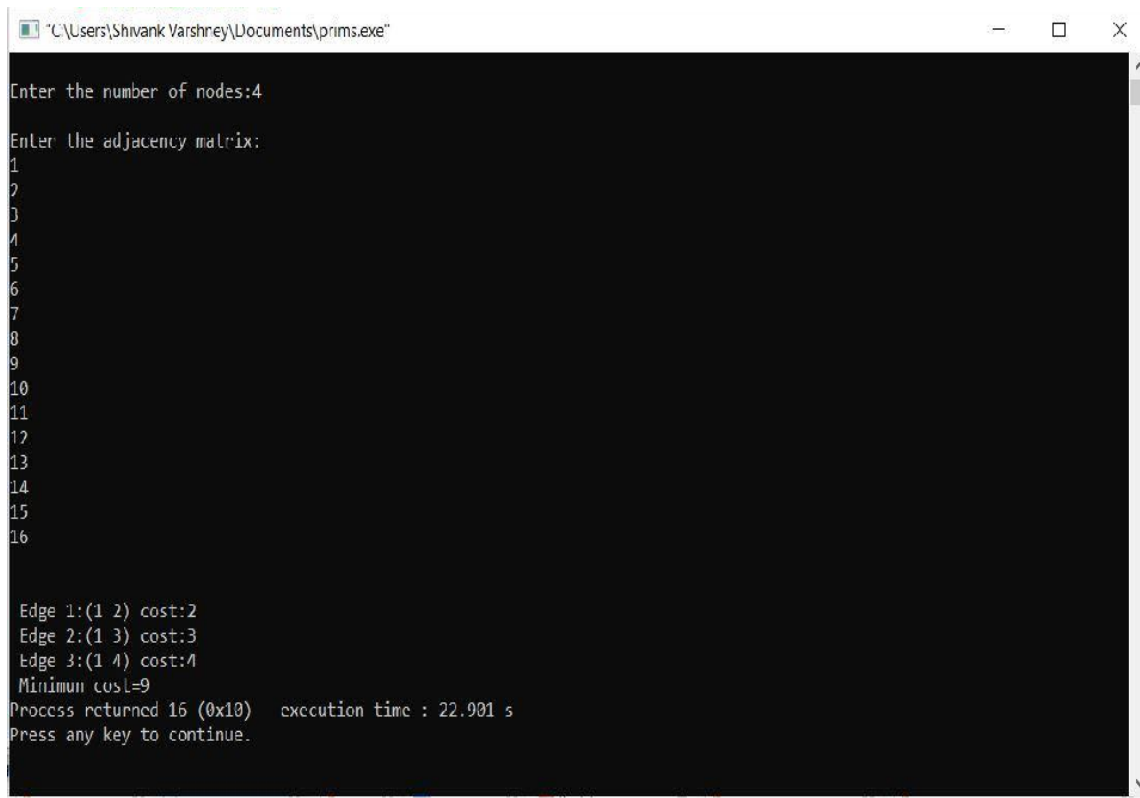
        visited[b] = 1;
    }

    cost[a][b] = cost[b][a] = 999;
}

printf("\n Minimun cost=%d", mincost);
return 0;
}

```

Output:



```

C:\Users\Shivank Varshney\Documents\prims.exe
Enter the number of nodes:4
Enter the adjacency matrix:
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

Edge 1:(1 2) cost:2
Edge 2:(1 3) cost:3
Edge 3:(1 4) cost:4
Minimun cost=9
Process returned 16 (0x10)   execution time : 22.901 s
Press any key to continue.

```

