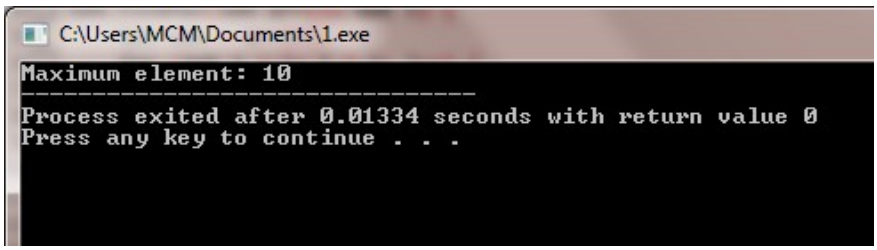
1)Removal of Recursion

Write a program to implement removal of recursion for-

a)Finding maximum from array.

```
#include <iostream>

using namespace std;

int findMax(int arr[], int n) {

    int max = arr[0];

    for (int i = 1; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

int main() {

    int arr[] = {5, 3, 9, 2, 8, 10};

    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum element: " << findMax(arr, n);

    return 0;

}
```

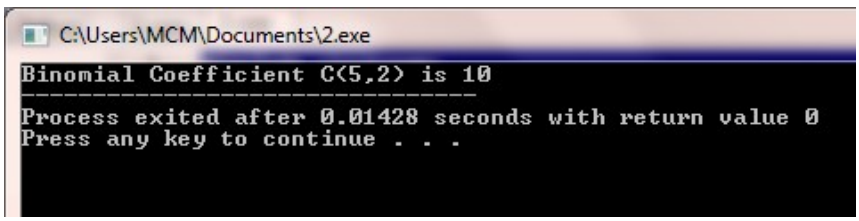
Output:



b)Binomial Coefficient B(n,m)=B(n-1,m-1)+B(n-1,m),B(n,n)=B(n,0)=1.

```
#include <iostream>

using namespace std;

int binomialCoeff(int m, int n) {
    int C[m + 1][n + 1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= min(i, n); j++) {
            if (j == 0 || j == i)
                C[i][j] = 1;
            else
                C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
        }
    }
    return C[m][n];
}

int main() {
    int m= 5, n = 2;
    cout << "Binomial Coefficient C(" << m << "," << n << ") is " << binomialCoeff(m, n);
    return 0;
}
```

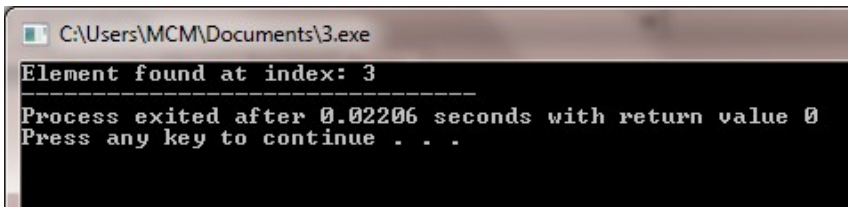
OUTPUT



c)Searching element from array.

```
#include <iostream>

using namespace std;

int linearSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;
    }
    return -1;
}

int main() {
    int arr[] = {5, 7, 2, 8, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 8;
    int result = linearSearch(arr, n, x);
    if (result != -1)
        cout << "Element found at index: " << result;
    else
        cout << "Element not found";
    return 0;
}
```

OUTPUT

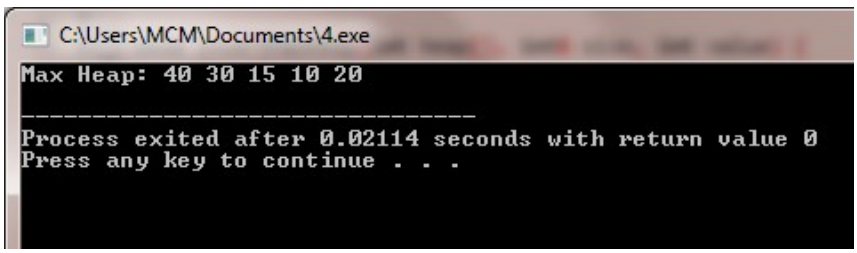


2)Elementary data structure-Tree

a)Write a program for creating max/min heap using INSERT.

```
#include <iostream>
using namespace std;
void insertMaxHeap(int heap[], int& size, int value) {
    size++;
    heap[size] = value;
    int i = size;
    while (i > 1 && heap[i / 2] < heap[i]) {
        swap(heap[i], heap[i / 2]);
        i = i / 2;
    }
}
void printHeap(int heap[], int size) {
    for (int i = 1; i <= size; i++)
        cout << heap[i] << " ";
    cout << endl;
}
int main() {
    int heap[100], size = 0;
    int elements[] = {10, 20, 15, 30, 40};
    for (int i = 0; i < 5; i++)
        insertMaxHeap(heap, size, elements[i]);
cout << "Max Heap: ";
    printHeap(heap, size);
    return 0;}
```

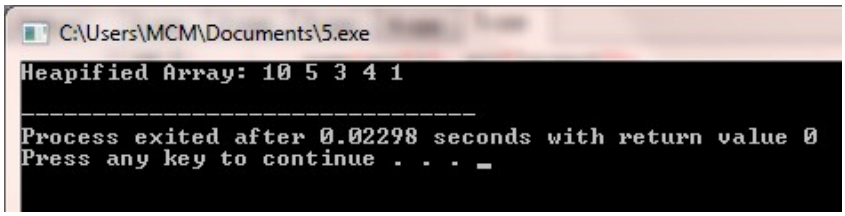
OUTPUT



b)Write a program for creating max/min heap using ADJUST/HEAPIFY.

```
#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 buildMaxHeap(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
}
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
}
int main() {
    int arr[] = {4, 10, 3, 5, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    buildMaxHeap(arr, n);
    cout << "Heapified Array: ";
    printArray(arr, n);
    return 0;
}
```

OUTPUT


C:\Users\MCM\Documents\5.exe
Heapified Array: 10 5 3 4 1

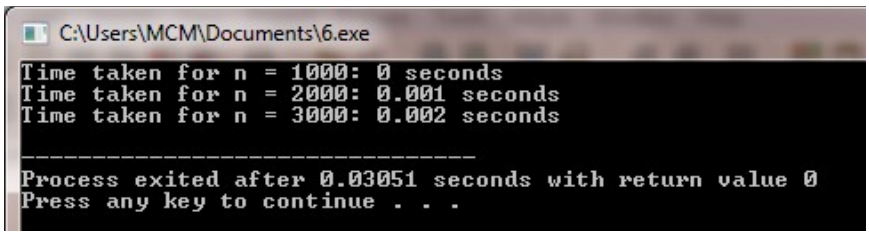
Process exited after 0.02298 seconds with return value 0
Press any key to continue . . .

c)Write a program for sorting given array in ascending /descending order with n=1000 ,2000,3000.find exact time of execution using heap sort.

```
#include <iostream>
#include <cstdlib>  // ? For rand() and srand()
#include <ctime>
using namespace std;
void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
            if (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    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 sizes[] = {1000, 2000, 3000};
            for (int s = 0; s < 3; s++) {
        int n = sizes[s];
        int* arr = new int[n];
```

```
        for (int i = 0; i < n; i++)
            arr[i] = rand();
                    clock_t start = clock();
        heapSort(arr, n);
        clock_t end = clock();
                    cout << "Time taken for n = " << n << ": "
            << (double)(end - start) / CLOCKS_PER_SEC << " seconds\n";


        delete[] arr;
    }
    return 0;
}
```

OUTPUT

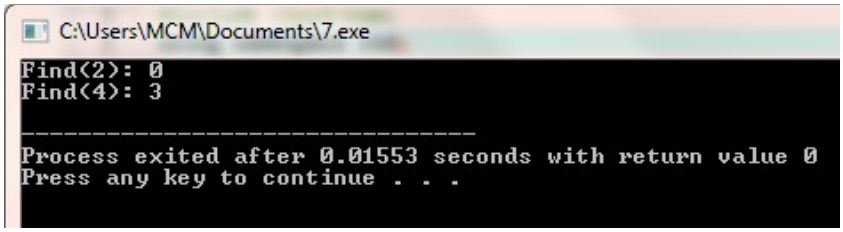


d)Write a program to implement Weighted UNION and Collapsing FIND operation.

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

const int MAX = 1000;

int parent[MAX], size[MAX];

void makeSet(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        size[i] = 1;
    }
}

int find(int x) {
    if (x != parent[x])
        parent[x] = find(parent[x]); // Path compression
    return parent[x];
}

void unionSets(int a, int b) {
    int rootA = find(a);
    int rootB = find(b);
    if (rootA != rootB) {
        if (size[rootA] < size[rootB])
            swap(rootA, rootB);
        parent[rootB] = rootA;
        size[rootA] += size[rootB];
    }
}

int main() {
    int n = 5;
    makeSet(n);
    unionSets(0, 1);
    unionSets(1, 2);
    unionSets(3, 4);
    cout << "Find(2): " << find(2) << endl;
```

```
    cout << "Find(4): " << find(4) << endl;

    return 0;

}
```

OUTPUT

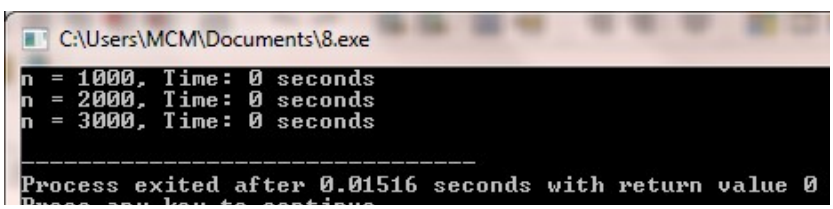


3)Divide and Conquer.

a)Write a program for searching element from given array using binary search for n=1000,2000,3000.Find exact time of execution .

```
#include <iostream>

#include <ctime>

using namespace std;

int binarySearch(int arr[], int l, int r, int x) {

   while (l <= r) {

      int mid = l + (r - l) / 2;

      if (arr[mid] == x) return mid;

      if (arr[mid] < x) l = mid + 1;

      else r = mid - 1; }

   return -1;}

int main() {

   int sizes[] = {1000, 2000, 3000};

   for (int s = 0; s < 3; s++) {

      int n = sizes[s];

      int* arr = new int[n];

      for (int i = 0; i < n; i++) arr[i] = i * 2;

      int key = arr[n - 1];  // Last element for worst case

      clock_t start = clock();

      int index = binarySearch(arr, 0, n - 1, key);

      clock_t end = clock();

      cout << "n = " << n << ", Time: "

          << (double)(end - start) / CLOCKS_PER_SEC << " seconds\n";

      delete[] arr;

   }

   return 0;}
```

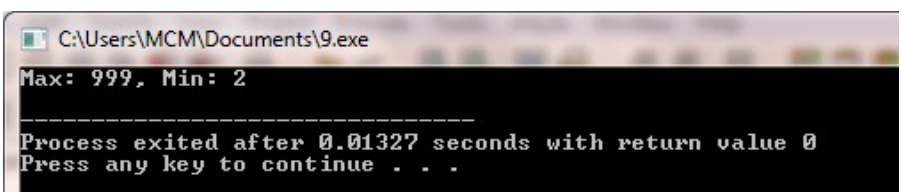
OUTPUT



b)Write a program to find maximum and minimum from given array using MAXMIN.

```
#include <iostream>
#include <utility>   // for std::pair
#include <algorithm> // for std::max and std::min
using namespace std;
pair<int, int> maxMin(int arr[], int low, int high) {
    int maxVal, minVal;
    if (low == high)
        return make_pair(arr[low], arr[low]);
    if (high == low + 1) {
        if (arr[low] > arr[high])
            return make_pair(arr[low], arr[high]);
        else
            return make_pair(arr[high], arr[low]);
    }
    int mid = (low + high) / 2;
    pair<int, int> left = maxMin(arr, low, mid);
    pair<int, int> right = maxMin(arr, mid + 1, high);
    maxVal = max(left.first, right.first);
    minVal = min(left.second, right.second);
    return make_pair(maxVal, minVal);}
int main() {
    int arr[] = {100, 200, 5, 2, 999, 21, 67};
    int n = sizeof(arr) / sizeof(arr[0]);
    pair<int, int> result = maxMin(arr, 0, n - 1);
    cout << "Max: " << result.first << ", Min: " << result.second << endl;
    return 0;
}
```

OUTPUT



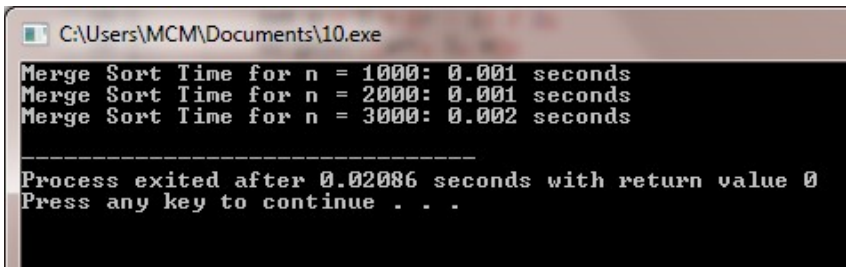
c)Write a program for sorting given array in ascending/descending order with n=1000,2000,3000 Find exact time of execution using –

d)Merge sort

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
void merge(int arr[], int beg, int mid, int end) {
int n1=mid-beg+1;
int n2=end-mid;
int leftarray[n1];
int rightarray[n2];
for(int i=0;i<n1;i++)
leftarray[i]=arr[beg+i];
for(int j=0;j<n2;j++)
rightarray[j]=arr[mid+1+j];
int i=0,j=0,k=beg;
while(i<n1 && j<n2){
if(leftarray[i]<=rightarray[j]){
            arr[k]=leftarray[i];
            i++;
            }
else{
     arr[k]=rightarray[j];
            j++;
}k++;
}while(i<n1){
arr[k]=leftarray[i];
            i++;
            k++;
  }
  while(j<n2){
            arr[k]=rightarray[j];
```

```
            j++;
            k++;
  }
}
void mergesort(int arr[],int beg,int end){
if(beg<end){
            int mid=(beg+end)/2;
            mergesort(arr,beg,mid);
            mergesort(arr,mid+1,end);
            merge(arr,beg,mid,end);
            }
}
int main() {
  int sizes[] = {1000, 2000, 3000};
  for (int s = 0; s < 3; s++) {
  int n = sizes[s];
   int* arr = new int[n];
 for (int i = 0; i < n; i++) arr[i] = rand() % 10000;
 clock_t start = clock();
 mergesort(arr, 0, n - 1);
 clock_t end = clock();
 cout << "Merge Sort Time for n = " << n << ": "
          << (double)(end - start) / CLOCKS_PER_SEC << " seconds\n";
   delete[] arr;
    }
 return 0;
}
```

OUTPUT

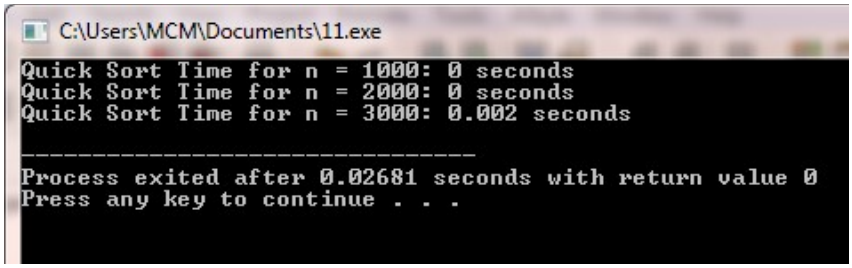


e)Quick sort

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int partition(int arr[], int low, int high) {
    int pivot = arr[high], i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int sizes[] = {1000, 2000, 3000};
    for (int s = 0; s < 3; s++) {
        int n = sizes[s];
        int* arr = new int[n];
        for (int i = 0; i < n; i++) arr[i] = rand() % 10000;
        clock_t start = clock();
        quickSort(arr, 0, n - 1);
        clock_t end = clock();
        cout << "Quick Sort Time for n = " << n << ": "
```

```
            << (double)(end - start) / CLOCKS_PER_SEC << " seconds\n";

        delete[] arr;

    }

    return 0;

}
```

OUTPUT

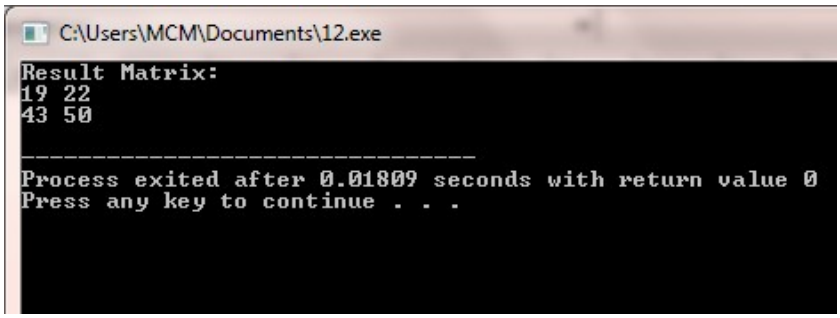


f)Write a program for matrix multiplication using Strassen's Matrix Multiplication.

```
#include <iostream>
using namespace std;
void add(int A[2][2], int B[2][2], int C[2][2]) {
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            C[i][j] = A[i][j] + B[i][j];
}
void subtract(int A[2][2], int B[2][2], int C[2][2]) {
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            C[i][j] = A[i][j] - B[i][j];
}
void strassen(int A[2][2], int B[2][2], int C[2][2]) {
    int M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
    int M2 = (A[1][0] + A[1][1]) * B[0][0];
    int M3 = A[0][0] * (B[0][1] - B[1][1]);
    int M4 = A[1][1] * (B[1][0] - B[0][0]);
    int M5 = (A[0][0] + A[0][1]) * B[1][1];
    int M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]);
    int M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);
    C[0][0] = M1 + M4 - M5 + M7;
    C[0][1] = M3 + M5;
    C[1][0] = M2 + M4;
    C[1][1] = M1 - M2 + M3 + M6;
}
int main() {
    int A[2][2] = {{1, 2}, {3, 4}};
    int B[2][2] = {{5, 6}, {7, 8}};
    int C[2][2];
    strassen(A, B, C);
    cout << "Result Matrix:\n";
    for (int i = 0; i < 2; i++) {
```

```
        for (int j = 0; j < 2; j++)
            cout << C[i][j] << " ";
        cout << endl;
    }
    return 0;
}
```

OUTPUT



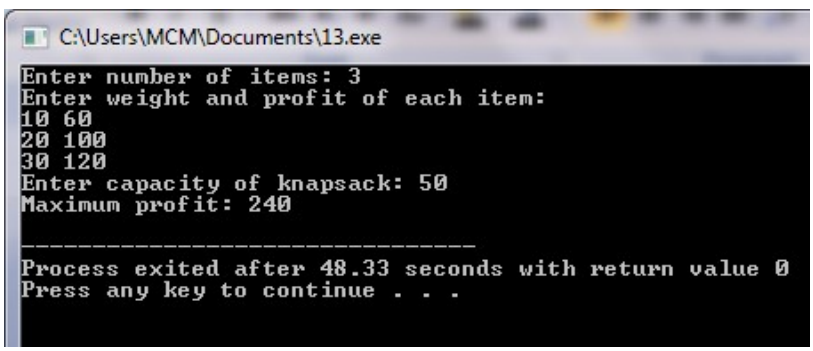
4) GREEDY ALGORITHM

a)Write a Program to find solution of Fractional Knapsack instance.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Structure to represent an item
struct Item {
    int weight;
    int profit;
    double ratio;
    // Constructor
    Item(int w, int p) {
        weight = w;
        profit = p;
        ratio = (double)p / w;
    }
};
// Comparison function to sort items by ratio
bool compare(Item a, Item b) {
    return a.ratio > b.ratio;
}
// Function to solve Fractional Knapsack
double fractionalKnapsack(vector<Item> items, int capacity) {
    // Sort items by value-to-weight ratio
    sort(items.begin(), items.end(), compare);
    double totalProfit = 0.0;
    int currWeight = 0;
    for (int i = 0; i < items.size(); ++i) {
        if (currWeight + items[i].weight <= capacity) {
            currWeight += items[i].weight;
            totalProfit += items[i].profit;
        } else {
```

```
            int remain = capacity - currWeight;
            totalProfit += items[i].ratio * remain;
            break;
        }
    }
    return totalProfit;
}
// Main function
int main() {
    int n, capacity;
    cout << "Enter number of items: ";
    cin >> n;
    vector<Item> items;
    cout << "Enter weight and profit of each item:\n";
    for (int i = 0; i < n; ++i) {
        int w, p;
        cin >> w >> p;
        items.push_back(Item(w, p));
    }
    cout << "Enter capacity of knapsack: ";
    cin >> capacity;
    double maxProfit = fractionalKnapsack(items, capacity);
    cout << "Maximum profit: " << maxProfit << endl;
    return 0;
}
```

OUTPUT

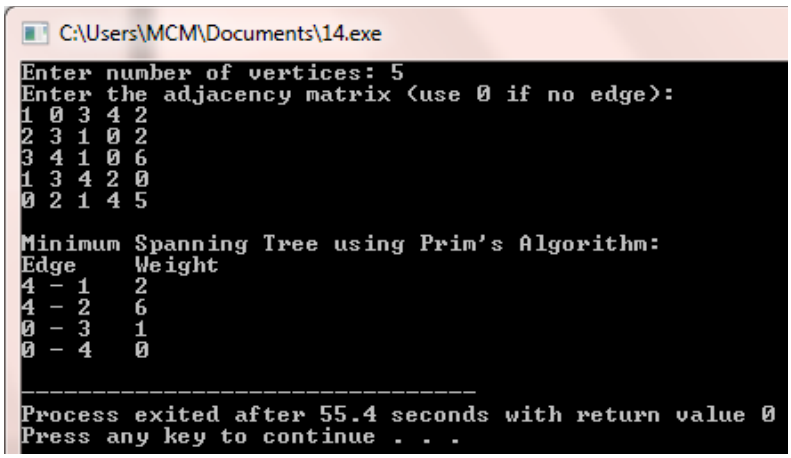


b)Write a program to find Minimum Spanning Tree using Prim's Algorithm.

```
#include <iostream>
#include <vector>
#include <climits> // for INT_MAX
using namespace std;
const int MAX = 100;
int findMinKey(int key[], bool mstSet[], int V) {
    int min = INT_MAX, minIndex;
    for (int v = 0; v < V; v++)
        if (!mstSet[v] && key[v] < min)
            min = key[v], minIndex = v;
    return minIndex;
}
void printMST(int parent[], int graph[MAX][MAX], int V) {
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << "\t" << graph[i][parent[i]] << "\n";
}
void primMST(int graph[MAX][MAX], int V) {
    int parent[V]; // Stores MST
    int key[V];    // Used to pick minimum weight edge
    bool mstSet[V]; // To represent included vertices
    // Initialize keys
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;
    key[0] = 0;       // Start from first vertex
    parent[0] = -1;   // Root of MST
    for (int count = 0; count < V - 1; count++) {
        int u = findMinKey(key, mstSet, V);
        mstSet[u] = true;
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
```

```
        }
    }
    printMST(parent, graph, V);
}
int main() {
    int V;
    cout << "Enter number of vertices: ";
    cin >> V;
    int graph[MAX][MAX];
    cout << "Enter the adjacency matrix (use 0 if no edge):\n";
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];
    cout << "\nMinimum Spanning Tree using Prim's Algorithm:\n";
    primMST(graph, V);
    return 0;
}
```

OUTPUT

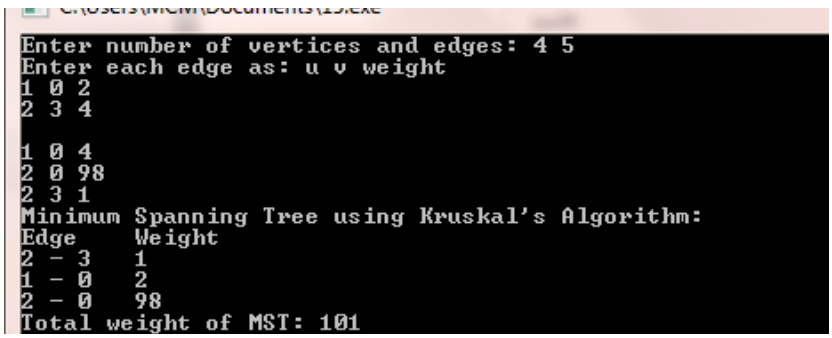


c)Write a program to find Minimum Spanning Tree using Kruskal's algorithm.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Edge {
    int u, v, weight;
};
bool compare(Edge a, Edge b) {
    return a.weight < b.weight;
}
class DisjointSet {
public:
    vector<int> parent;
    DisjointSet(int n) {
        parent.resize(n);
        for (int i = 0; i < n; ++i)
            parent[i] = i;
    }
    int find(int i) {
        if (parent[i] != i)
            parent[i] = find(parent[i]);
        return parent[i];
    }
    void unionSet(int u, int v) {
        int set_u = find(u);
        int set_v = find(v);
        if (set_u != set_v)
            parent[set_u] = set_v;
    }};
void kruskalMST(int V, vector<Edge>& edges) {
    sort(edges.begin(), edges.end(), compare);
    DisjointSet ds(V);
```

```
    vector<Edge> mst;
    int totalWeight = 0;
    for (int i = 0; i < edges.size(); ++i) {
        Edge e = edges[i];
        if (ds.find(e.u) != ds.find(e.v)) {
            mst.push_back(e);
            totalWeight += e.weight;
            ds.unionSet(e.u, e.v);
        } }
cout << "Minimum Spanning Tree using Kruskal's Algorithm:\n";
    cout << "Edge\tWeight\n";
    for (int i = 0; i < mst.size(); ++i)
        cout << mst[i].u << " - " << mst[i].v << "\t" << mst[i].weight << "\n";
    cout << "Total weight of MST: " << totalWeight << "\n";
}
int main() {
    int V, E;
    cout << "Enter number of vertices and edges: ";
    cin >> V >> E;
    vector<Edge> edges(E);
    cout << "Enter each edge as: u v weight\n";
    for (int i = 0; i < E; ++i)
        cin >> edges[i].u >> edges[i].v >> edges[i].weight;
kruskalMST(V, edges);
    return 0;
}
```

OUTPUT

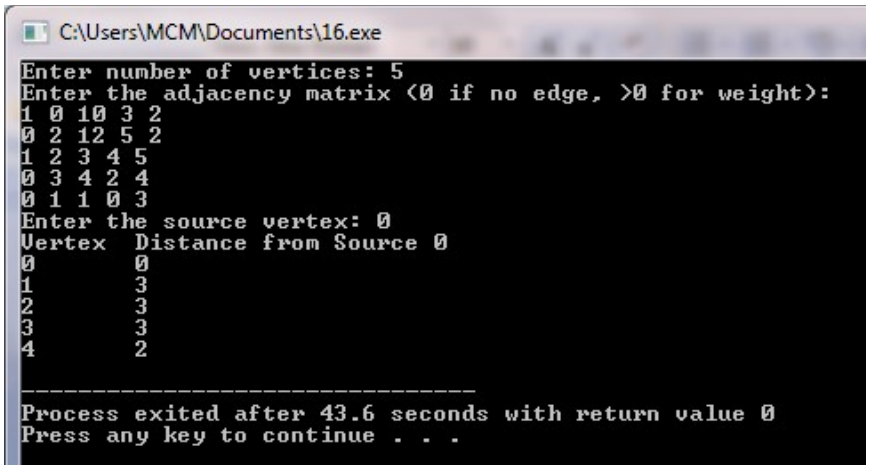


d)Write a program to find Single Source Shortest Path using Dijkstra's algorithm

```
#include <iostream>
#include <vector>
#include <limits>
using namespace std;
#define INF 99999  // Representing infinity
void dijkstra(int graph[10][10], int V, int src) {
    vector<int> dist(V, INF);      // Distance from source to each vertex
    vector<bool> visited(V, false); // Track visited vertices
    dist[src] = 0; // Distance to source is 0
    for (int count = 0; count < V - 1; ++count) {
        // Find the minimum distance vertex from the set of unvisited vertices
        int u = -1;
        int minDist = INF;
        for (int i = 0; i < V; ++i) {
            if (!visited[i] && dist[i] < minDist) {
                minDist = dist[i];
                u = i;
            }
        }
        if (u == -1) break; // No reachable vertex left
        visited[u] = true;
        // Update distances to neighboring vertices
        for (int v = 0; v < V; ++v) {
            if (!visited[v] && graph[u][v] && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
        }
    }
    // Print the result
    cout << "Vertex\tDistance from Source " << src << endl;
    for (int i = 0; i < V; ++i)
        cout << i << "\t" << dist[i] << endl;}
```

```
int main() {
    int V;
    cout << "Enter number of vertices: ";
    cin >> V;
    int graph[10][10];
    cout << "Enter the adjacency matrix (0 if no edge, >0 for weight):" << endl;
    for (int i = 0; i < V; ++i)
        for (int j = 0; j < V; ++j)
            cin >> graph[i][j];
    int src;
    cout << "Enter the source vertex: ";
    cin >> src;
    dijkstra(graph, V, src);
    return 0;
}
```

OUTPUT

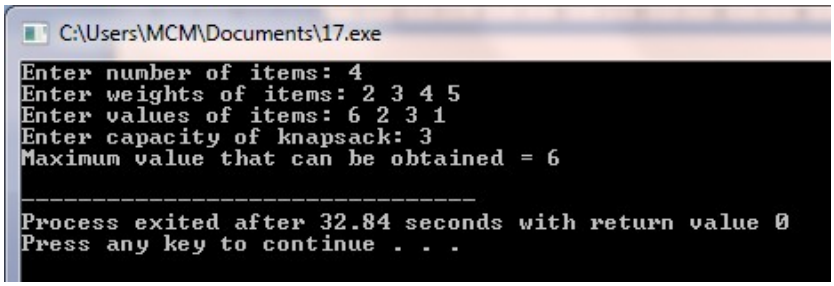


5)Dynamic Programming

a)Write a program to find solution of Knapsack Instance (0/1).

```
#include <iostream>
#include <vector>
using namespace std;
int knapsack(int W, vector<int>& weights, vector<int>& values, int n) {
    vector<vector<int> > dp(n + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= W; w++) {
            if (weights[i - 1] <= w) {
                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];
            }
        }
    }
    return dp[n][W];
}
int main() {
    int n, W;
    cout << "Enter number of items: ";
    cin >> n;
    vector<int> weights(n), values(n);
    cout << "Enter weights of items: ";
    for (int i = 0; i < n; i++) cin >> weights[i];
    cout << "Enter values of items: ";
    for (int i = 0; i < n; i++) cin >> values[i];
    cout << "Enter capacity of knapsack: ";
    cin >> W;
    int maxValue = knapsack(W, weights, values, n);
    cout << "Maximum value that can be obtained = " << maxValue << endl;
    return 0;}
```

OUTPUT


C:\Users\MCM\Documents\17.exe
Enter number of items: 4
Enter weights of items: 2 3 4 5
Enter values of items: 6 2 3 1
Enter capacity of knapsack: 3
Maximum value that can be obtained = 6

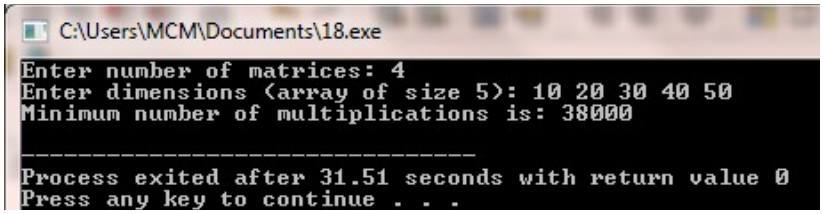
Process exited after 32.84 seconds with return value 0
Press any key to continue . . .

b)Write a program to find solution of matrix chain multiplication.

```
#include <iostream>
#include <limits.h>
using namespace std;
// Function to compute minimum number of multiplications
int matrixChainMultiplication(int p[], int n) {
    int m[n][n];
  // m[i][j] = Minimum number of multiplications needed to compute A[i]A[i+1]...A[j] = A[i..j]
    for (int i = 1; i < n; i++) {
        m[i][i] = 0; // cost is 0 when multiplying one matrix
    }
    // l is chain length
    for (int l = 2; l < n; l++) {
        for (int i = 1; i < n - l + 1; i++) {
            int j = i + l - 1;
            m[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                int q = m[i][k] + m[k + 1][j] + p[i - 1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }
    return m[1][n - 1];
}
int main() {
    int n;
    cout << "Enter number of matrices: ";
    cin >> n;
    int p[n + 1];
    cout << "Enter dimensions (array of size " << n + 1 << "): ";
    for (int i = 0; i <= n; i++) {
        cin >> p[i];
```

```
    }

    int result = matrixChainMultiplication(p, n + 1);

    cout << "Minimum number of multiplications is: " << result << endl;

    return 0;

}
```

OUTPUT

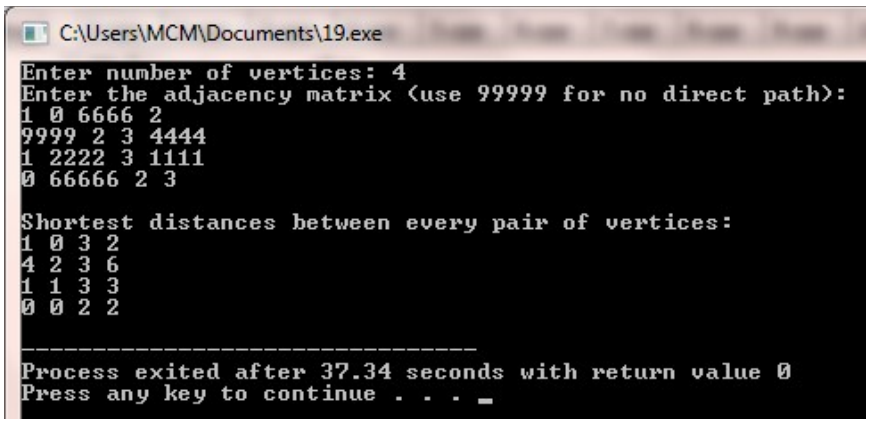


c)Write a program to find shortest path using all pair shortest path algorithm.

```
#include <iostream>
using namespace std;
#define INF 99999
#define V 100 // Maximum number of vertices
void floydWarshall(int graph[V][V], int n) {
    int dist[V][V];
    // Initialize the solution matrix same as input graph matrix
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = graph[i][j];
    // Floyd-Warshall Algorithm
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                // Update the distance if going through vertex k is shorter
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    // Print the shortest distance matrix
    cout << "\nShortest distances between every pair of vertices:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (dist[i][j] == INF)
                cout << "INF ";
            else
                cout << dist[i][j] << " ";
        }
        cout << endl;
    }
}
```

```
int main() {
    int n;
    cout << "Enter number of vertices: ";
    cin >> n;
    int graph[V][V];
    cout << "Enter the adjacency matrix (use " << INF << " for no direct path):\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> graph[i][j];
    floydWarshall(graph, n);
    return 0;
}
```

OUTPUT

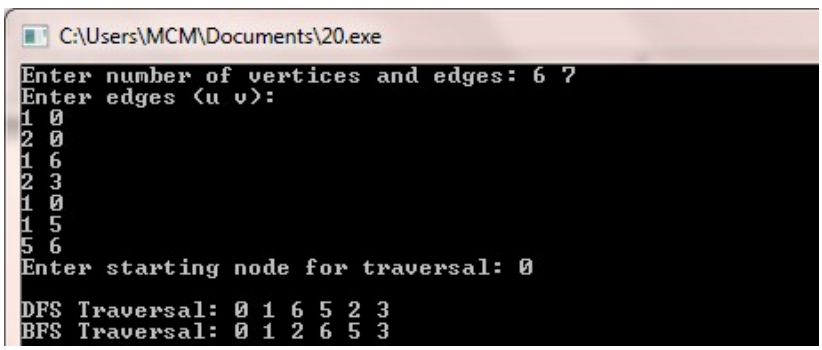


d)Write a program to Traverse Graph-Depth First Search,Breadth First Search.

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
#define MAX 100
vector<int> adj[MAX];
bool visited[MAX];
// Depth First Search (DFS)
void DFS(int node) {
    visited[node] = true;
    cout << node << " ";
    for (int i = 0; i < adj[node].size(); i++) {
        int neighbor = adj[node][i];
        if (!visited[neighbor])
            DFS(neighbor);
    }
}
// Breadth First Search (BFS)
void BFS(int start) {
    for (int i = 0; i < MAX; i++)
        visited[i] = false;
    queue<int> q;
    visited[start] = true;
    q.push(start);
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        for (int i = 0; i < adj[node].size(); i++) {
            int neighbor = adj[node][i];
            if (!visited[neighbor]) {
                visited[neighbor] = true;
```

```
            q.push(neighbor);
        }
    }
  }
}
int main() {
  int vertices, edges, u, v, start;
  cout << "Enter number of vertices and edges: ";
  cin >> vertices >> edges;
  cout << "Enter edges (u v):\n";
  for (int i = 0; i < edges; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u); // for undirected graph
  }
  cout << "Enter starting node for traversal: ";
  cin >> start;
  // DFS
  for (int i = 0; i < MAX; i++)
    visited[i] = false;
  cout << "\nDFS Traversal: ";
  DFS(start);
  // BFS
  cout << "\nBFS Traversal: ";
  BFS(start);
  return 0;}
```

OUTPUT



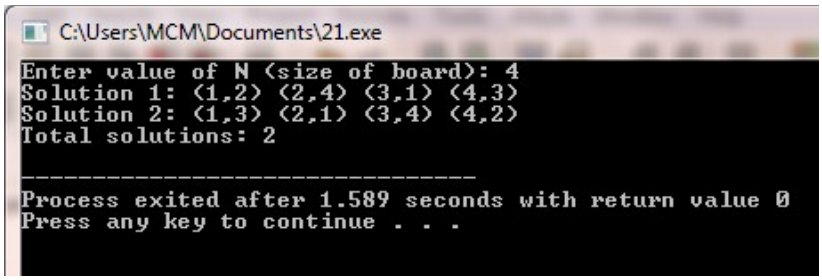
6)BackTracking

a)Write a program to find all solutions for N-Queen problem using backtracking.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
#define MAX 20
int board[MAX], count = 0;
// Function to check if position is safe
bool isSafe(int row, int col, int n) {
    for (int i = 1; i < row; i++) {
        // Check column and diagonals
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
            return false;
    }
    return true;
}
// Recursive backtracking function
void solveNQueens(int row, int n) {
    if (row > n) {
        count++;
        cout << "Solution " << count << ": ";
        for (int i = 1; i <= n; i++)
            cout << "(" << i << "," << board[i] << ") ";
        cout << endl;
        return;
    }
    for (int col = 1; col <= n; col++) {
        if (isSafe(row, col, n)) {
            board[row] = col;
            solveNQueens(row + 1, n);
        }
    }}
```

```
int main() {
    int n;
    cout << "Enter value of N (size of board): ";
    cin >> n;
 if (n < 1 || n > MAX) {
       cout << "Invalid board size. Use N between 1 and " << MAX << ".\n";
        return 0;
    }
    solveNQueens(1, n);
    if (count == 0)
        cout << "No solution exists for N = " << n << endl;
    else
        cout << "Total solutions: " << count << endl;
    return 0;
}
```

OUTPUT



b)Write a program for Graph Coloring using backtracking.

```
#include <iostream>
using namespace std;
#define MAX 20
int graph[MAX][MAX]; // Adjacency matrix
int color[MAX];      // Color assigned to each vertex
int V, M;            // V = number of vertices, M = number of colors
// Check if the current color assignment is safe
bool isSafe(int v, int c) {
    for (int i = 0; i < V; i++) {
        if (graph[v][i] == 1 && color[i] == c)
            return false;
    }
    return true;
}
// Backtracking function to assign colors
bool graphColoring(int v) {
    if (v == V)  // All vertices are assigned
        return true;
    for (int c = 1; c <= M; c++) {
        if (isSafe(v, c)) {
            color[v] = c;
            if (graphColoring(v + 1))
                return true;
            // Backtrack
            color[v] = 0;
        }
    }
    return false;
}
int main() {
    cout << "Enter number of vertices: ";
    cin >> V;
```

```
    cout << "Enter number of colors: ";
    cin >> M;
    cout << "Enter adjacency matrix (" << V << "x" << V << "):\n";
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            cin >> graph[i][j];
    for (int i = 0; i < V; i++)
        color[i] = 0;  // Initialize all colors to 0
    if (graphColoring(0)) {
        cout << "Solution Exists. Assigned colors:\n";
        for (int i = 0; i < V; i++)
            cout << "Vertex " << i << " --> Color " << color[i] << endl;
    } else {
        cout << "No solution exists with " << M << " colors.\n";
    }
    return 0;}
```

OUTPUT