To write a C program to implement **Prim's Minimum Spanning Tree (MST)** algorithm using an **edge list**, selecting minimum-weight edges to form a spanning tree.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define V 5 // Number of vertices in the graph
// Structure to represent an edge
struct Edge {
  int src, dest, weight;
};
// Function to compare edges for qsort
int compare(const void *a, const void *b) {
  return ((struct Edge *)a)->weight - ((struct Edge *)b)->weight;
}

// Function to find the vertex with the minimum key value
/* Algorithm Step:
  1. Set min = INF.
  2. Scan all vertices; pick vertex 'v' not in MST with smallest key[v].
  3. Return that vertex index.
*/
int minKey(int key[], int mstSet[]) {
  int min = INT_MAX, minIndex;
  for (int v = 0; v < V; v++) {
    if (mstSet[v] == 0 && key[v] < min) {
      min = key[v];
      minIndex = v;
    }
  }
  return minIndex;
}

// Function to implement Prim's algorithm using an edge list
/* PRIM'S ALGORITHM:
  1. Initialize all keys = INF, mstSet = 0.
  2. Set key[0] = 0 (start vertex), parent[0] = -1.
  3. Repeat V-1 times:
     a. Pick u = vertex with minimum key not yet in MST.
     b. Mark u as included in MST.
     c. For each edge touching u:
        - If the other vertex is not in MST
          AND edge weight < key[vertex],
          update key and parent.
  4. Print parent[] and key[] as MST edges.
*/
void primMST(struct Edge edges[], int edgeCount) {
  int parent[V]; // MST parent array
  int key[V];   // Minimum edge weights
  int mstSet[V]; // Included in MST or not

  // Step 1: Initialize all
  for (int i = 0; i < V; i++) {
    key[i] = INT_MAX;
    mstSet[i] = 0;
  }

  // Step 2: Start from vertex 0
  key[0] = 0;
  parent[0] = -1;

  // Step 3: Build MST
  for (int count = 0; count < V - 1; count++) {
    int u = minKey(key, mstSet); // Pick min key vertex
    mstSet[u] = 1;           // Include in MST

    // Step 4: Update keys of adjacent vertices
    for (int i = 0; i < edgeCount; i++) {
      if (edges[i].src == u && mstSet[edges[i].dest] == 0) {
        if (edges[i].weight < key[edges[i].dest]) {
          parent[edges[i].dest] = u;
          key[edges[i].dest] = edges[i].weight;
        }
      } else if (edges[i].dest == u && mstSet[edges[i].src] == 0) {
        if (edges[i].weight < key[edges[i].src]) {
          parent[edges[i].src] = u;
          key[edges[i].src] = edges[i].weight;
        }
      }
    }
  }
  // Step 5: Output MST
  printf("Edge \tWeight\n");
  for (int i = 1; i < V; i++)
    printf("%d - %d \t%d\n", parent[i], i, key[i]);
}
int main() {
  // Example graph represented as an edge list
  struct Edge edges[] = {
    {0, 1, 2},
    {0, 3, 6},
    {1, 2, 3},
    {1, 3, 8},
    {1, 4, 5},
    {2, 4, 7},
    {3, 4, 9}
  };
  int edgeCount = sizeof(edges) / sizeof(edges[0]);
  // Call the Prim's algorithm function
  primMST(edges, edgeCount);
  return 0;
}
```

To write a C program to implement **Quick Sort** using the **divide-and-conquer** approach with partitioning.

```
#include <stdio.h>

// Function to swap two elements
/* Step: Swap two numbers using a temporary variable */
void swap(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

// Function to partition the array
/* PARTITION ALGORITHM:
  1. Choose last element as pivot.
  2. Set i = low-1 (tracks smaller elements).
  3. For each j from low to high-1:
     - If arr[j] <= pivot: increase i and swap(arr[i], arr[j]).
  4. Place pivot in correct position by swapping arr[i+1] with arr[high].
  5. Return pivot index (i+1).
*/
int partition(int arr[], int low, int high) {
  int pivot = arr[high];
  int i = (low - 1);

  for (int j = low; j <= high - 1; j++) {
    if (arr[j] <= pivot) {
      i++;
      swap(&arr[i], &arr[j]);
    }
  }
  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}

// Function that implements Quick Sort
/* QUICK SORT ALGORITHM:
  1. If low < high:
     a. Partition array → get pivot index pi.
     b. Recursively sort left subarray (low to pi-1).
     c. Recursively sort right subarray (pi+1 to high).
*/
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);
  }
}

// Function to print an array
/* Step: Traverse and print each element */
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
}

// Main function to test the Quick Sort implementation
/* Steps:
  1. Initialize array.
  2. Print unsorted array.
  3. Call quickSort().
  4. Print sorted array.
*/
int main() {
  int arr[] = {10, 7, 8, 9, 1, 5};
  int n = sizeof(arr) / sizeof(arr[0]);
  printf("Unsorted array: \n");
  printArray(arr, n);

  quickSort(arr, 0, n - 1);

  printf("Sorted array: \n");
  printArray(arr, n);
  return 0;
}
```

To write a C program to multiply two 2×2 matrices using **Strassen's Matrix Multiplication Algorithm**.

```
#include<stdio.h>
int main() {
  int a[2][2],b[2][2],c[2][2],i,j;
  int m1,m2,m3,m4,m5,m6,m7;

  printf("enter the 4 elements of the 1st matix:");
  for(i=0;i<2;i++){
    for(j=0;j<2;j++){
      scanf("%d",&a[i][j]);
    }
  }

  printf("enter the 4 elements of the 2nd matix:\n");
  for(i=0;i<2;i++){
    for(j=0;j<2;j++){
      scanf("%d",&b[i][j]);
    }
  }

  printf("\n the first matrix is \n");
  for(i=0;i<2;i++){
    for(j=0;j<2;j++){
      printf("%d\t",a[i][j]);
    }
    printf("\n");
  }

  printf("\n the second matix is \n");
  for(i=0;i<2;i++){
    for(j=0;j<2;j++){
      printf("%d\t",b[i][j]);
    }
    printf("\n");
  }

  /* STRASSEN'S ALGORITHM STEPS:
    1. Compute the 7 Strassen products:
      m1=(a11+a22)(b11+b22)
      m2=(a21+a22)b11
      m3=a11(b12-b22)
      m4=a22(b21-b11)
      m5=(a11+a12)b22
      m6=(a21-a11)(b11+b12)
      m7=(a12-a11)(b21+b22)
    2. Compute result matrix:
      c11 = m1 + m4 - m5 + m7
      c12 = m3 + m5
      c21 = m2 + m4
      c22 = m1 - m2 + m3 + m6
  */

  m1=(a[0][0]+a[1][1])*(b[0][0]+b[1][1]);
  m2=(a[1][0]+a[1][1])*b[0][0];
  m3=a[0][0]*(b[0][1]-b[1][1]);
  m4=a[1][1]*(b[1][0]-b[0][0]);
  m5=(a[0][0]+a[0][1])*b[1][1];
  m6=(a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
  m7=(a[0][1]-a[0][0])*(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;

  printf("\n the strassen matix is \n");
  for(i=0;i<2;i++){
    printf("\n");
    for(j=0;j<2;j++){
      printf("%d\t",c[i][j]);
    }
  }

  return 0;
}
```

To write a C program to search an element in a sorted array using the **recursive Binary Search algorithm**.

```c
#include <stdio.h>

// Recursive binary search function
/* BINARY SEARCH ALGORITHM:
  1. If left <= right:
     a. Compute mid = (left + right)/2.
     b. If arr[mid] == target → return mid.
     c. If arr[mid] < target → search right half.
     d. Else → search left half.
  2. If element not found → return -1.
*/
int binarySearchRecursive(int arr[], int left, int right, int target)
{
  if (left <= right) {
    int mid = left + (right - left) / 2;

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

    if (arr[mid] < target)
      return binarySearchRecursive(arr, mid + 1, right, target);

    return binarySearchRecursive(arr, left, mid - 1, target);
  }

  return -1; // Not found
}

int main() {
  /* STEPS:
    1. Declare sorted array.
    2. Call binarySearchRecursive().
    3. Print result.
  */
  int arr[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
  int size = sizeof(arr) / sizeof(arr[0]);
  int target = 12;

  int result = binarySearchRecursive(arr, 0, size - 1, target);

  if (result != -1)
    printf("Element %d found at index %d\n", target, result);
  else
    printf("Element %d not found in the array\n", target);

  return 0;
}
```

To write a C program to compute the **all-pairs shortest paths** in a weighted graph using the **Floyd-Warshall algorithm**.

```c
#include <stdio.h>
#include <limits.h>   // Include the limits.h header for INT_MAX

// Number of vertices in the graph
#define V 4

// Function to print the solution matrix
void printSolution(int dist[][V]);

// Implementation of the Floyd-Warshall algorithm
/* FLOYD–WARSHALL ALGORITHM:
  1. Copy graph[][] to dist[][] (initial distances).
  2. For each vertex k (intermediate):
      For each pair (i, j):
        If path i → k → j is shorter than i → j:
          Update dist[i][j].
  3. Print final shortest-path matrix.
*/
void floydWarshall(int graph[][V]) {
  int dist[V][V];

  // Step 1: Initialize distance matrix with graph values
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      dist[i][j] = graph[i][j];

  // Step 2: Try every vertex as intermediate (k loop)
  for (int k = 0; k < V; k++) {
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {

        /* Check if going through k gives a shorter path:
          dist[i][k] + dist[k][j] < dist[i][j]
          (Also check for INF to avoid overflow)
        */
        if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX
          && dist[i][k] + dist[k][j] < dist[i][j])
          dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }

  // Step 3: Print final result
  printSolution(dist);
}

// Function to print the solution matrix
/* Step: Print each cell → print INF if no path exists */
void printSolution(int dist[][V]) {
  printf("The following matrix shows the shortest distances between every pair of vertices:\n");
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (dist[i][j] == INT_MAX)
        printf("INF ");
      else
        printf("%d  ", dist[i][j]);
    }
    printf("\n");
  }
}

int main() {
  /* STEPS:
    1. Define adjacency matrix (use INT_MAX for no edge).
    2. Call floydWarshall().
  */
  int graph[V][V] = {
    {0, 3, INT_MAX, 5},
    {2, 0, INT_MAX, 4},
    {INT_MAX, INT_MAX, 0, 1},
    {INT_MAX, INT_MAX, INT_MAX, 0}
  };

  floydWarshall(graph);

  return 0;
}
```

To write a C program to sort an array using the **Heapsort algorithm** by building a max-heap and repeatedly extracting the maximum element.

```c
#include <stdio.h>

// Function to heapify a subtree rooted at node i of the array arr[] of size n
/* HEAPIFY ALGORITHM:
  1. Set largest = i.
  2. Compute left = 2*i+1, right = 2*i+2.
  3. If left child > root → largest = left.
  4. If right child > largest → largest = right.
  5. If largest != i:
     a. Swap arr[i] and arr[largest].
     b. Recursively heapify the affected subtree.
*/
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) {
    int temp = arr[i];
    arr[i] = arr[largest];
    arr[largest] = temp;

    heapify(arr, n, largest);
  }
}

// Main function to perform heapsort on an array
/* HEAPSORT ALGORITHM:
  1. Build a max-heap by calling heapify() from last non-leaf to root.
  2. For i = n-1 down to 1:
     a. Swap arr[0] (max) with arr[i].
     b. Reduce heap size (i).
     c. Call heapify() on root to restore max-heap.
*/
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--) {
    int temp = arr[0];
    arr[0] = arr[i];
    arr[i] = temp;

    heapify(arr, i, 0);
  }
}

// Function to print an array
/* Step: Print all elements */
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

int main() {
  /* STEPS:
    1. Initialize array.
    2. Print before sorting.
    3. Call heapsort().
    4. Print sorted array.
  */
  int arr[] = {12, 11, 13, 5, 6, 7};
  int n = sizeof(arr) / sizeof(arr[0]);

  printf("Original array: ");
  printArray(arr, n);

  heapsort(arr, n);

  printf("Sorted array: ");
  printArray(arr, n);

  return 0;
}
```