```
void merge(int arr[], int left, int mid, int right) {
  int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];
   for (int i = 0; i < n1; i++) L[i] = arr[left + i];
   for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];
   int i = 0, j = 0, k = left;
   while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
   while (i < n1) arr[k++] = L[i++];
   while (j < n2) arr[k++] = R[j++];}
void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);   }}
int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);
   mergeSort(arr, 0, n - 1);
   for (int i = 0; i < n; i++) cout << arr[i] << " ";
}
```

```
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) 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 arr[] = {10, 7, 8, 9, 1, 5};
   int n = sizeof(arr) / sizeof(arr[0]);
quickSort(arr, 0, n - 1);
for (int i = 0; i < n; i++) cout << arr[i] << " ";
}
```

```
void heapify(int arr[], int n, int i) {
   int largest = i, left = 2 * i + 1, 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 arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);
   heapSort(arr, n);
   for (int i = 0; i < n; i++) cout << arr[i] << " ";}
```

```
class PriorityQueue {
   int arr[100];
   int size;
public:
   PriorityQueue() : size(0) {}
   void push(int val) {
      int i = size++;
      while (i > 0 && arr[(i - 1) / 2] < val) {
         arr[i] = arr[(i - 1) / 2];
         i = (i - 1) / 2;}
      arr[i] = val;}
   void pop() {
      int root = arr[--size], i = 0, child;
      while ((child = 2 * i + 1) < size) {
         if (child + 1 < size && arr[child + 1] > arr[child]) child++;
         if (root >= arr[child]) break;
```

```
int binarySearch(int arr[], int low, int high, int key) {
   if (low > high)
      return -1; // Key not found
   int mid = low + (high - low) / 2;
   if (arr[mid] == key)
      return mid; // Key found
   else if (arr[mid] < key)
      return binarySearch(arr, mid + 1, high, key); // Right half
   else
      return binarySearch(arr, low, mid - 1, key); // Left half}
int main() {
   int arr[] = {1, 3, 5, 7, 9};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key = 5;
   int result = binarySearch(arr, 0, n - 1, key);
   if (result != -1)
      cout << "Element found at index " << result << endl;
   else
      cout << "Element not found" << endl;}
```

```
#include <climits>
#define V 5  // Number of vertices in the graph
int minDistance(int dist[], bool sptSet[]) {
   int min = INT_MAX, minIndex;   for (int v = 0; v < V; v++) {
      if (!sptSet[v] && dist[v] <= min) {  min = dist[v];
         minIndex = v;}} return minIndex;} void printSolution(int
dist[]) {cout << "Vertex \t Distance from Source\n";
   for (int i = 0; i < V; i++)  cout << i << " \t " << dist[i] << "\n";}
void dijkstra(int graph[V][V], int src) {int dist[V];     bool sptSet[V];
   for (int i = 0; i < V; i++) {dist[i] = INT_MAX;
      sptSet[i] = false; }dist[src] = 0;  for (int count = 0; count < V -
1; count++) {  int u = minDistance(dist, sptSet);
      sptSet[u] = true; for (int v = 0; v < V; v++) {if (!sptSet[v] &&
graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] <
dist[v]) {dist[v] = dist[u] + graph[u][v];}}} printSolution(dist);}
int main() { int graph[V][V] = {
      {0, 10, 20, 0, 0},
      {10, 0, 30, 50, 10},
      {20, 30, 0, 20, 0},
      {0, 50, 20, 0, 60},
      {0, 10, 0, 60, 0}}; dijkstra(graph, 0);  // Starting from vertex 0}
```

```
#include <algorithm>
struct Item {
   int value, weight;};
bool cmp(Item a, Item b) {
   return (double)a.value / a.weight > (double)b.value /
b.weight;}
double fractionalKnapsack(int W, Item arr[], int n) {
   sort(arr, arr + n, cmp);
   double totalValue = 0.0;
   for (int i = 0; i < n; i++) {
      if (W >= arr[i].weight) {
         W -= arr[i].weight;
         totalValue += arr[i].value;
      } else {
         totalValue += arr[i].value * ((double)W / arr[i].weight);
         break;}}
   return totalValue;}
int main() {
   Item arr[] = {{60, 10}, {100, 20}, {120, 30}};
   int W = 50, n = sizeof(arr) / sizeof(arr[0]);
   cout << "Maximum value in the knapsack: " <<
fractionalKnapsack(W, arr, n) << endl;}
```

```
#include <climits>
#define V 5
int minKey(int key[], bool mstSet[]) {
   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[V][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[V][V]) {  int parent[V]  int key[V    bool
mstSet[V];    for (int i = 0; i < V; i++) {       key[i] = INT_MAX;
      mstSet[i] = false   }    key[0] = 0;    parent[0] = -1;
   for (int count = 0; count < V - 1; count++) {int u = minKey(key,
mstSet);   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);}
int main() { int graph[V][V] = {
      {0, 2, 0, 6, 0},
      {2, 0, 3, 8, 5},
      {0, 3, 0, 0, 7},
      {6, 8, 0, 0, 9},
      {0, 5, 7, 9, 0} };primMST(graph);}
```

```
#include <algorithm> struct Edge {
   int src, dest, weight;};
struct Subset {int parent, rank;};int find(Subset subsets[], int i) {
   if (subsets[i].parent != i)  subsets[i].parent = find(subsets, subsets[i].parent);
   return subsets[i].parent;}void Union(Subset subsets[], int x, int y) {
   int rootX = find(subsets, x);int rootY = find(subsets, y);
   if (subsets[rootX].rank < subsets[rootY].rank)  subsets[rootX].parent = rootY;
   else if (subsets[rootX].rank > subsets[rootY].rank)subsets[rootY].parent = rootX;
   else {subsets[rootY].parent = rootX;subsets[rootX].rank++;}}
bool cmp(Edge a, Edge b) {return a.weight < b.weight;}
void kruskalMST(Edge edges[], int V, int E) {
   sort(edges, edges + E, cmp);Subset* subsets = new Subset[V];for (int v = 0; v < V;
v++) {  subsets[v].parent = v;  subsets[v].rank = 0;}
   Edge* result = new Edge[V - 1];int e = 0, i = 0;
while (e < V - 1 && i < E) {  Edge nextEdge = edges[i++];
int x = find(subsets, nextEdge.src);       int y = find(subsets, nextEdge.dest);
      if (x != y) {          result[e++] = nextEdge;         Union(subsets, x, y);    }}
   cout << "Edge \tWeight\n";for (int i = 0; i < e; i++)  cout << result[i].src << " - " <<
result[i].dest << "\t" << result[i].weight << "\n";
   delete[] subsets;    delete[] result;}
int main()  int V = 4; // Number of vertice   int E = 5; // Number of edges
   Edge edges[] = {{0, 1, 10},{0, 2, 6},{0, 3, 5},{1, 3, 15},{2, 3, 4}};
   kruskalMST(edges, V, E);}
```

```
arr[i] = arr[child];
         i = child  }
      arr[i] = root }
   int top() { return arr[0]; }
   bool empty() { return size == 0; }};
int main() {
   PriorityQueue pq;
   pq.push(10);
   pq.push(30);
   pq.push(20);
   while (!pq.empty()) {
      cout << pq.top() << " ";
      pq.pop();
   }
   return 0;
}
```

```
#include <iostream>
#include <algorithm>
int knapsack(int W, int wt[], int val[], int n) {
   int dp[n + 1][W + 1];
   for (int i = 0; i <= n; i++) {
      for (int w = 0; w <= W; w++) {
         if (i == 0 || w == 0)
            dp[i][w] = 0;
         else if (wt[i - 1] <= w)
            dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i -
1][w]);
         else
            dp[i][w] = dp[i - 1][w];       }  }
   return dp[n][W];}
int main() {
   int W = 50;
   int wt[] = {10, 20, 30};
   int val[] = {60, 100, 120};
   int n = sizeof(val) / sizeof(val[0]);
   cout << knapsack(W, wt, val, n);}
```

```
#include <queue>
struct Node {char ch;int freq;    Node *left, *right;
   Node(char c, int f) : ch(c), freq(f), left(NULL), right(NULL) {}};
struct Compare {   bool operator()(Node* l, Node* r) {       return
l->freq > r->freq;   };
void printCodes(Node* root, string str) {
   if (!root) return; if (root->ch != '$') cout << root->ch << ": " << str
<< endl;    printCodes(root->left, str + "0");
   printCodes(root->right, str + "1");}
void huffman(char arr[], int freq[], int n) {
priority_queue<Node*, vector<Node*>, Compare> pq;
   for (int i = 0; i < n; i++) pq.push(new Node(arr[i], freq[i]));
   while (pq.size() > 1) {
      Node* left = pq.top(); pq.pop();
      Node* right = pq.top(); pq.pop();
      Node* top = new Node('$', left->freq + right->freq);
      top->left = left;
      top->right = right;
      pq.push(top) }
   printCodes(pq.top(), "");}
int main() {
   char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
   int freq[] = {5, 9, 12, 13, 16, 45};
   int n = sizeof(arr) / sizeof(arr[0]);
   huffman(arr, freq, n);}
```