Quicksort –

```
#include <stdio.h>

// Swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Partition function (pivot = first element)
int partition(int arr[], int start, int end) {
    int pivot = arr[start];
    int count = 0;

    // Count elements <= pivot
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot) {
            count++;
        }
    }

    int pivotIndex = start + count;
    swap(&arr[pivotIndex], &arr[start]);

    int i = start, j = end;
    while (i < pivotIndex && j > pivotIndex) {
        while (arr[i] <= pivot) i++;
        while (arr[j] > pivot) j--;
        if (i < pivotIndex && j > pivotIndex) {
            swap(&arr[i], &arr[j]);
```

```
            i++;
            j--;
        }
    }

    return pivotIndex;
}

// Quick Sort function
void quicksort(int arr[], int start, int end) {
    if (start >= end)
        return;

    int pivotIndex = partition(arr, start, end);

    quicksort(arr, start, pivotIndex - 1);
    quicksort(arr, pivotIndex + 1, end);
}

// Driver code
int main() {
    int arr[] = {2, 5, 7, 7, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quicksort(arr, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
```

```
    return 0;
}
```

Mergesort –

```
#include <stdio.h>
void merge(int arr[],int l,int mid,int r)
{
    int n1=mid-l+1;
    int n2=r-mid;
    int left[n1];
    int right[n2];
    int i,j,k;
    for(int i=0;i<n1;i++)
    {
        left[i]=arr[l+i];
    }
    for(int j=0;j<n2;j++)
    {
        right[j]=arr[mid+1+j];
    }
    i=0;
    j=0;
    k=l;
    while(i<n1 && j<n2)
    {
        if(left[i]<=right[j])
        {
            arr[k]=left[i];
            k++;
            i++;
        }
```

```
        else
        {
            arr[k]=right[j];
            k++;
            j++;
        }
    }
        while(i<n1)
        {
            arr[k]=left[i];
            k++;
            i++;
        }
        while(j<n2)
        {
            arr[k]=right[j];
            k++;
            j++;
        }

}
void mergesort(int arr[],int l,int r)
{
    if(l>=r) return;
    int mid=(l+r)/2;
    mergesort(arr,l,mid);
    mergesort(arr,mid+1,r);
    merge(arr,l,mid,r);

}
int main()
```

```
{
   int arr[]={2,3,4,6,4,7};
   int n=sizeof(arr)/sizeof(arr[0]);
   for(int i=0;i<n;i++)
   {
      printf("%d ",arr[i]);
   }
   printf("\n");
   mergesort(arr,0,n-1);
   for(int i=0;i<n;i++)
   {
      printf("%d ",arr[i]);
   }
}
```

Greedy Knapsack -

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

struct Item {
   int value;
   int weight;
   double ratio;
};

void selectionsort(struct Item items[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (items[j].ratio > items[i].ratio) {
            struct Item temp = items[j];
            items[j] = items[i];
```

```
            items[i] = temp;
        }
    }
  }
}

double fractionalKnapsack(int val[], int wt[], int n, int capacity) {
  struct Item items[n];

  // Step 1: Fill items array
  for (int i = 0; i < n; i++) {
    items[i].value = val[i];
    items[i].weight = wt[i];
    items[i].ratio = (double) val[i] / wt[i];
  }

  // Step 2: Sort by ratio (descending)
  selectionsort(items, n);

  double totalvalue = 0.0;
  int remaining = capacity;

  printf("\nItems selected:\n");
  printf("Value\tWeight\tRatio\t\tTaken\n");
  printf("----------------------------------------\n");

  // Step 3: Take items greedily
  for (int i = 0; i < n && remaining > 0; i++) {
    if (items[i].weight <= remaining) {
      // Take the full item
      totalvalue += items[i].value;
```

```
            remaining -= items[i].weight;
            printf("%d\t%d\t%.2f\t\tFull\n", items[i].value, items[i].weight, items[i].ratio);
        } else {
            // Take fractional part
            double fraction = (double) remaining / items[i].weight;
            totalvalue += items[i].value * fraction;
            printf("%d\t%d\t%.2f\t\t%.2f part\n", items[i].value, items[i].weight, items[i].ratio, fraction);
            remaining = 0;
            break;
        }
    }

    printf("----------------------------------------\n");
    printf("Total value in knapsack = %.2f\n", totalvalue);
    return totalvalue;
}

int main(void) {
    int n = 3, capacity = 50;
    int val[] = {60, 100, 120};
    int wt[]  = {10, 20, 30};

    double ans = fractionalKnapsack(val, wt, n, capacity);
    printf("\nExpected output: 240.00\n");
    return 0;
}
```

Huffman Coding.cpp –

```
#include <iostream>
#include <queue>
#include <vector>
```

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

// A node of the Huffman tree
struct Node {
    char ch;         // Character
    int freq;        // Frequency
    Node *left, *right; // Child pointers

    Node(char c, int f) {
        ch = c;
        freq = f;
        left = right = nullptr;
    }
};

// Comparator for priority queue (min-heap)
struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq; // smaller freq = higher priority
    }
};

// Recursive function to print Huffman Codes
void printCodes(Node* root, string code) {
    if (!root)
        return;

    // Leaf node → print character and its code
    if (!root->left && !root->right) {
        cout << root->ch << " : " << code << endl;
```

```
        return;
    }

    // Go left → add '0'
    printCodes(root->left, code + "0");
    // Go right → add '1'
    printCodes(root->right, code + "1");
}

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

    vector<char> chars(n);
    vector<int> freq(n);

    cout << "Enter characters: ";
    for (int i = 0; i < n; i++)
        cin >> chars[i];

    cout << "Enter corresponding frequencies: ";
    for (int i = 0; i < n; i++)
        cin >> freq[i];

    // Step 1: Create a min-heap (priority queue)
    priority_queue<Node*, vector<Node*>, Compare> pq;

    // Step 2: Push all characters as nodes into heap
    for (int i = 0; i < n; i++) {
        pq.push(new Node(chars[i], freq[i]));
```

```
    }

    // Step 3: Combine two smallest until one remains
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();

        // Create a new internal node with frequency = sum of two
        Node* newNode = new Node('$', left->freq + right->freq);
        newNode->left = left;
        newNode->right = right;

        pq.push(newNode);
    }

    // Step 4: The remaining node is the root
    Node* root = pq.top();

    cout << "\nHuffman Codes:\n";
    printCodes(root, "");

    return 0;
}
```

Kruskals –

```
#include <stdio.h>

#define MAX 30

struct Edge {
   int u, v, w;
};

int parent[MAX];

int findParent(int i) {
   while (parent[i] != i)
      i = parent[i];
   return i;
}

void unionSet(int i, int j) {
   int a = findParent(i);
   int b = findParent(j);
   parent[a] = b;
}

void kruskal(int n, int e, struct Edge edges[]) {
   int i, j;
   struct Edge temp;
   int totalCost = 0, count = 0;

   // Sort edges by weight (Bubble Sort)
   for (i = 0; i < e - 1; i++) {
      for (j = 0; j < e - i - 1; j++) {
```

```
            if (edges[j].w > edges[j + 1].w) {
                temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }

    // Initialize parent array
    for (i = 0; i < n; i++)
        parent[i] = i;

    printf("\nEdges in the Minimum Spanning Tree (Kruskal's):\n");
    for (i = 0; i < e && count < n - 1; i++) {
        int a = findParent(edges[i].u);
        int b = findParent(edges[i].v);

        if (a != b) {
            printf("%d -- %d == %d\n", edges[i].u, edges[i].v, edges[i].w);
            totalCost += edges[i].w;
            unionSet(a, b);
            count++;
        }
    }

    printf("Total weight of MST = %d\n", totalCost);
}

int main() {
    struct Edge edges[MAX];
    int n, e, i;
```

```
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);

    printf("Enter edges (u v w):\n");
    for (i = 0; i < e; i++)
        scanf("%d%d%d", &edges[i].u, &edges[i].v, &edges[i].w);

    kruskal(n, e, edges);

    return 0;
}
```

Prims –

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

#define MAX 20

void prims(int cost[MAX][MAX], int n) {
    int visited[MAX] = {0};
    int totalCost = 0, edges = 0;

    visited[0] = 1; // Start from vertex 0

    printf("\nEdges in the Minimum Spanning Tree (Prim's):\n");

    while (edges < n - 1) {
        int min = INT_MAX, u = -1, v = -1;
```

```
        for (int i = 0; i < n; i++) {
            if (visited[i]) {
                for (int j = 0; j < n; j++) {
                    if (!visited[j] && cost[i][j] && cost[i][j] < min) {
                        min = cost[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
        }

        if (u != -1 && v != -1) {
            printf("%d -- %d == %d\n", u, v, min);
            totalCost += min;
            visited[v] = 1;
            edges++;
        }
    }

    printf("Total weight of MST = %d\n", totalCost);
}

int main() {
    int cost[MAX][MAX], n;

    printf("Enter number of vertices: ");
    scanf("%d", &n);

    printf("Enter cost adjacency matrix (0 if no edge):\n");
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0)
                cost[i][j] = INT_MAX; // No edge
        }
    }

    prims(cost, n);

    return 0;
}
```

Dp_knapsack –

```
#include <stdio.h>

// Function to find the greater of two numbers
int findMax(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

// Function to solve 0/1 Knapsack using Dynamic Programming
int knapSack(int capacity, int weight[], int value[], int n) {
    int dp[n + 1][capacity + 1];

    // Step 1: Build DP table
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
```

```
        // Base case: 0 items or 0 capacity
        if (i == 0 || w == 0) {
            dp[i][w] = 0;
        }
        // If current item's weight <= current capacity
        else if (weight[i - 1] <= w) {
            int includeItem = value[i - 1] + dp[i - 1][w - weight[i - 1]];
            int excludeItem = dp[i - 1][w];
            dp[i][w] = findMax(includeItem, excludeItem);
        }
        // If item can't be included (too heavy)
        else {
            dp[i][w] = dp[i - 1][w];
        }
    }
}

// Step 2: Backtrack to find items included
int result = dp[n][capacity];
int remainingCapacity = capacity;

printf("\nItems included in the bag:\n");
for (int i = n; i > 0 && result > 0; i--) {
    if (result != dp[i - 1][remainingCapacity]) {
        printf("Item %d -> Weight = %d, Value = %d\n", i, weight[i - 1], value[i - 1]);
        result = result - value[i - 1];
        remainingCapacity = remainingCapacity - weight[i - 1];
    }
}

// Step 3: Print DP Table (for understanding)
```

```
    printf("\nDP Table:\n");
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            printf("%3d ", dp[i][w]);
        }
        printf("\n");
    }

    return dp[n][capacity];
}

// Main Function
int main() {
    int n, capacity;

    printf("Enter number of items: ");
    scanf("%d", &n);

    int value[n], weight[n];

    printf("Enter values (profits): ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &value[i]);
    }

    printf("Enter weights: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &weight[i]);
    }

    printf("Enter knapsack capacity: ");
```

```
    scanf("%d", &capacity);

    int maxValue = knapSack(capacity, weight, value, n);
    printf("\nMaximum value that can be obtained = %d\n", maxValue);

    return 0;
}
```

Dp_tsp –

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

#define MAX 10
#define INF 999999

int n;
int cost[MAX][MAX];
int dp[1 << MAX][MAX];    // DP table: (bitmask, current city)
int nextCity[1 << MAX][MAX]; // To reconstruct path
int visited_all;

/* Recursive function to solve TSP using DP + Bitmasking */
int tsp(int mask, int pos) {
    if (mask == visited_all)
        return cost[pos][0]; // Return cost to start city

    if (dp[mask][pos] != -1)
        return dp[mask][pos];

    int ans = INF;
    int chosenCity = -1;
```

```
    for (int city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) { // if city not visited
            int newAns = cost[pos][city] + tsp(mask | (1 << city), city);
            if (newAns < ans) {
                ans = newAns;
                chosenCity = city;
            }
        }
    }

    nextCity[mask][pos] = chosenCity; // Store next city for this state
    dp[mask][pos] = ans;
    return ans;
}

/* Function to print the path */
void printPath() {
    int mask = 1, pos = 0;

    printf("\nPath: ");
    printf("0 "); // start from city 0

    while (1) {
        int city = nextCity[mask][pos];
        if (city == -1)
            break;
        printf("-> %d ", city);
        mask |= (1 << city);
        pos = city;
    }
```

```
    printf("-> 0\n"); // return to start
}

int main() {
    printf("Enter number of cities: ");
    scanf("%d", &n);

    printf("Enter cost matrix (0 if same city):\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &cost[i][j]);
        }
    }

    visited_all = (1 << n) - 1;

    // initialize DP arrays with -1
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            dp[i][j] = -1;
            nextCity[i][j] = -1;
        }
    }

    int answer = tsp(1, 0); // start from city 0 with mask 1 (only city 0 visited)

    printf("\nMinimum cost of travelling all cities = %d\n", answer);
    printPath();

    return 0;
}
```

Backtracking_hamilton –

ASS – 6 A)

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

#define MAX 10   // maximum number of vertices (you can increase if needed)

// Function to check if vertex v can be added at index 'pos' in the Hamiltonian Cycle
bool isSafe(int v, int graph[MAX][MAX], int path[], int pos, int V) {
   // 1. Check if current vertex is adjacent to the previous vertex
   if (graph[path[pos - 1]][v] == 0)
      return false;

   // 2. Check if vertex has already been included
   for (int i = 0; i < pos; i++)
      if (path[i] == v)
         return false;

   return true;
}

// Recursive utility function to solve the Hamiltonian Cycle problem
bool hamCycleUtil(int graph[MAX][MAX], int path[], int pos, int V) {
   // Base case: all vertices are included
   if (pos == V) {
      // Check if the last vertex connects back to the first vertex
      return (graph[path[pos - 1]][path[0]] == 1);
   }
```

```
    // Try different vertices as the next candidate
    for (int v = 1; v < V; v++) {
        if (isSafe(v, graph, path, pos, V)) {
            path[pos] = v;

            // Recur to construct the rest of the path
            if (hamCycleUtil(graph, path, pos + 1, V))
                return true;

            // Backtrack if adding v doesn't lead to a solution
            path[pos] = -1;
        }
    }

    return false;
}

// Function to solve the Hamiltonian Cycle problem using Backtracking
bool hamCycle(int graph[MAX][MAX], int V) {
    int path[MAX];

    // Initialize path[] as -1
    for (int i = 0; i < V; i++)
        path[i] = -1;

    // Start from vertex 0
    path[0] = 0;

    if (!hamCycleUtil(graph, path, 1, V)) {
        printf("\nNo Hamiltonian Cycle exists for the given graph.\n");
        return false;
```

```
    }

    // Print the Hamiltonian Cycle
    printf("\nHamiltonian Cycle found:\n");
    for (int i = 0; i < V; i++)
        printf("%d ", path[i]);
    printf("%d\n", path[0]); // to complete the cycle

    return true;
}

// Driver code
int main() {
    int V;
    int graph[MAX][MAX];

    printf("Enter number of vertices (max %d): ", MAX);
    scanf("%d", &V);

    printf("\nEnter the adjacency matrix (%d x %d):\n", V, V);
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("\nAdjacency Matrix:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++)
            printf("%d ", graph[i][j]);
        printf("\n");
```

```
    }

    hamCycle(graph, V);

    return 0;
}
```

Backtracking_nqueen –

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

#define MAX 10  // maximum board size

// Function to print the chessboard
void printSolution(int board[MAX][MAX], int N) {
    printf("\nOne of the possible solutions:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%c ", board[i][j] ? 'Q' : '.'); // Q = Queen, . = Empty
        }
        printf("\n");
    }
}

// Check if placing a queen at board[row][col] is safe
bool isSafe(int board[MAX][MAX], int row, int col, int N) {
    int i, j;

    // Check column above
    for (i = 0; i < row; i++)
        if (board[i][col])
```

```
            return false;

    // Check upper left diagonal
    for (i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;

    // Check upper right diagonal
    for (i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++)
        if (board[i][j])
            return false;

    return true;
}

// Recursive function to solve N-Queens problem
bool solveNQUtil(int board[MAX][MAX], int row, int N) {
    // Base case: if all queens are placed
    if (row == N) {
        printSolution(board, N);
        return true;
    }

    bool res = false; // to check if at least one solution exists

    // Try placing queen in all columns of current row
    for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col, N)) {
            board[row][col] = 1; // place queen

            // Recur to place the rest
```

```
            res = solveNQUtil(board, row + 1, N) || res;

            // Backtrack (remove queen)
            board[row][col] = 0;
        }
    }

    return res;
}

// Main function to initiate solving process
void solveNQ(int N) {
    int board[MAX][MAX] = {0};

    if (!solveNQUtil(board, 0, N))
        printf("\nNo solution exists for N = %d\n", N);
}

// Driver code
int main() {
    int N;
    printf("Enter the number of queens (N): ");
    scanf("%d", &N);
    solveNQ(N);
    return 0;
}
```

Branchandbound_tsp.cpp –

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

```
int n;                  // number of cities
int cost[10][10];       // cost matrix
int final_path[11];      // final tour
bool visited[10];       // visited cities
int final_cost = INT_MAX; // minimum cost

// Copy current path to final_path
void copyToFinal(int curr_path[]) {
   for (int i = 0; i < n; i++)
      final_path[i] = curr_path[i];
   final_path[n] = curr_path[0]; // return to start
}

// Get first minimum edge cost from city i
int firstMin(int i) {
   int min = INT_MAX;
   for (int k = 0; k < n; k++)
      if (cost[i][k] < min)
         min = cost[i][k];
   return min;
}

// Get second minimum edge cost from city i
int secondMin(int i) {
   int first = INT_MAX, second = INT_MAX;
   for (int j = 0; j < n; j++) {
      if (cost[i][j] < first) {
         second = first;
         first = cost[i][j];
      } else if (cost[i][j] < second) {
```

```
            second = cost[i][j];
        }
    }
    return second;
}

// Recursive TSP function
void TSP(int curr_path[], int curr_cost, int level) {
    if (level == n) {
        // Return to starting city
        if (cost[curr_path[level-1]][curr_path[0]] != 0) {
            int total_cost = curr_cost + cost[curr_path[level-1]][curr_path[0]];
            if (total_cost < final_cost) {
                final_cost = total_cost;
                copyToFinal(curr_path);
            }
        }
        return;
    }

    for (int i = 0; i < n; i++) {
        if (!visited[i] && cost[curr_path[level-1]][i] != 0) {

            // Calculate lower bound for pruning
            int bound = curr_cost;
            if (level == 1)
                bound += firstMin(curr_path[0]) + firstMin(i);
            else
                bound += secondMin(curr_path[level-1]) + firstMin(i);

            if (bound + cost[curr_path[level-1]][i] >= final_cost)
```

```
            continue; // prune

        // Make move
        curr_path[level] = i;
        visited[i] = true;

        TSP(curr_path, curr_cost + cost[curr_path[level-1]][i], level + 1);

        // Backtrack
        visited[i] = false;
        curr_path[level] = -1;
      }
    }
}

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

    cout << "Enter " << n << "x" << n << " cost matrix (0 for same city, space separated):\n";
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        cin >> cost[i][j];
      }
    }

    int curr_path[11];
    for (int i = 0; i <= n; i++) {
      curr_path[i] = -1;
      visited[i] = false;
      final_path[i] = -1;
```

```
    }

    // Start from city 0
    curr_path[0] = 0;
    visited[0] = true;

    TSP(curr_path, 0, 1);

    // Output result
    cout << "\nMinimum cost: " << final_cost << endl;
    cout << "Path: ";
    for (int i = 0; i < n; i++) {
        cout << final_path[i];
        if (i < n-1) cout << " -> ";
    }
    cout << " -> 0" << endl;

    return 0;
}
```