**EXP 1:-**

**Insertion Sort**: each element is 'inserted' into its correct position within a sorted subarray. It works by iterating over the array, selecting one element at a time, and moving it to its correct position within the already sorted part of the array. Efficient for small datasets or nearly sorted arrays, but TC grows quadratically with the size of the input for worst-case scenarios.

The algorithm iterates over the array, *starting from the second element* (index 1) to the last element. Each element compared with the elements before it, large elements 1 position to right until it finds correct position for current element. Repeated until entire array is sorted.

**void insertion\_sort(int A[], int size){  
 int i, j, key;**

// Outer loop iterating over each element of the array, starting from the second element (index 1)

**for(j = 1; j < size; j++)**

**{**

// Store the current element in the variable 'key'

**key = A[j];**

// Initialize variable 'i' to the index before the current element

**i = j - 1;**

// Inner loop to find correct position for current element within sorted portion of array

**while(i >= 0 && A[i] > key){**

// Shift elements greater than 'key' one position to the right

**A[i + 1] = A[i];**

// Move to the previous element to continue comparing

**i--;**

**}**

// Place the 'key' element in its correct position

**A[i + 1] = key;**

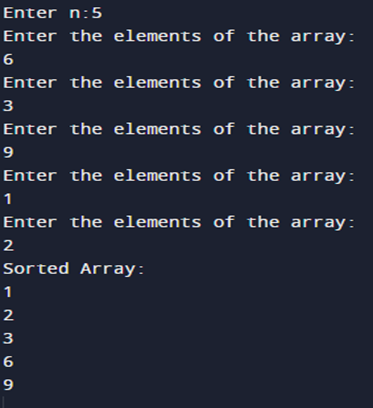
**}**

**}**

**Worst Case:** Descending order, performs max no. of comparisons and element shifts. **O()**.

**Best Case:** Already sorted, each element compared only once with its predecessor, **O(n)**.

**Average Case:** Balanced distribution of input elements. **O()**



**Quadratic TC**

**EXP 2:-**

**Quick Sort**: divide-and-conquer strategy. Selects a pivot element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.

It recursively sorts the sub-arrays until the entire array is sorted.  
**Worst Case:** Pivot selection consistently results in unbalanced partitions, leading to **O()** due to the partitioning process requiring linear time in each recursive call.

**Best Case:** Balanced partitions, leading to a time complexity of **O(n\*log n)** as array is divided into roughly equal halves in each recursive call.

**Average Case:** **O(n\*log n)**, when pivot selection results in partitions of roughly equal size.

**void swap(int\* a, int\* b) {**

**int temp = \*a;** // Store the value of 'a' in 'temp'

**\*a = \*b;** // Assign the value of 'b' to 'a'

**\*b = temp;** // Assign the stored value of 'a' (in 'temp') to 'b'

**}**

**int partition(int arr[], int low, int high) {**

**int pivot = arr[high];** // Select the pivot element as the last element of the array

**int i = low - 1;**  // Initialize index of smaller element to be one less than thelow index

// Iterate through the array

**for (int j = low; j < high; j++) {**

**if (arr[j] <= pivot) {**

**i++;** // Increment the index of the smaller element

**swap(&arr[i], &arr[j]);**  // Swap arr[i] and arr[j]

**}**

**}**

**swap(&arr[i + 1], &arr[high]);** // placing the pivot in its correct position

**return i + 1;**  // Return the index of the pivot element

**}**

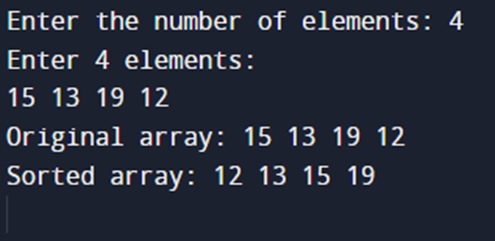
**void quickSort(int arr[], int low, int high) {**

**if (low < high) {**

**int pi = partition(arr, low, high);**  // Partition the array and get the pivot index

// Recursively sort the elements before and after the pivot

**quickSort(arr, low, pi - 1);**

**quickSort(arr, pi + 1, high);**

**}**

**}**

**void printArray(int arr[], int size) {**

**for (int i = 0; i < size; i++) {**

**printf("%d ", arr[i]);**

**}**

**}**

**int main() {**

//……………………..skipped……………………….

**printArray(arr, n);**  // Print the original array

**quickSort(arr, 0, n - 1);**  // Sort the array using QuickSort

**printArray(arr, n);** // Print the sorted array

**return 0;**

**}**

**EXP 3:-**

**Dijkstra's Algorithm:** find the shortest path from a source vertex to all other vertices in a weighted graph. Iteratively selecting the vertex with the shortest distance from the source among the unvisited vertices and relaxing (updating) distances to its neighboring vertices.  
**Graph:** Structure consisting of a set of vertices (nodes) and a set of edges (connections) that establish reln b/w vertices. Represented using an adjacency matrix or adjacency list.

**Shortest Path:** The shortest path b/w 2 vertices in a graph is path with the min total weight.

1. Initialize an array to store shortest distance from source vertex to each vertex in the graph. Set the distance to the source vertex as 0 and all other distances as infinity.
2. Select unvisited vertex with min dist from source vertex. Initially, this will be source vertex itself.
3. For selected vertex, examine its neighboring vertices (adjacent vertices) and calculate their tentative distances from source vertex through selected vertex. If newly calculated dist is shorter than the current known distance, update the distance.
4. After considering all neighbors of the selected vertex, mark the vertex as visited. A visited vertex will not be considered again during the algorithm.
5. Repeat steps 2-4 until all vertices have been visited or until the shortest path to the target vertex has been found.

**#include <stdio.h>**

**#include <stdlib.h>**

**#define MAX\_VERTICES 100**

**#define INFINITY 1000000**

// Structure to represent a graph

**typedef struct {**

**int vertices;** // Number of vertices in the graph

**int graph[MAX\_VERTICES][MAX\_VERTICES];** // Adjacency matrix

**} Graph;**

**void dijkstra(Graph\* g, int start) {** //Graph\* g declares a pointer g that can hold memory address of a Graph

**int dist[MAX\_VERTICES];**  // Array to store shortest distances from source vertex

**int pred[MAX\_VERTICES];**  // “ store predecessors of vertices in shortest paths

**int visited[MAX\_VERTICES];** // Array to track visited vertices

**for (int i = 0; i < g->vertices; ++i) {**

**dist[i] = INFINITY;**  // Set initial distances to infinity

**pred[i] = -1;** // Set initial predecessors to -1 (undefined)

**visited[i] = 0;** // Mark all vertices as unvisited

**}**

**dist[start] = 0;** // Distance from source to itself is 0

**for (int count = 0; count < g->vertices - 1; ++count) {**

**int u = -1;**

// Select the unvisited vertex with the minimum distance

**for (int i = 0; i < g->vertices; ++i) {**

**if (!visited[i] && (u == -1 || dist[i] < dist[u])) {**

**u = i;**

**}**

**}**

**visited[u] = 1;** // Mark the selected vertex as visited

// Update distances to its neighbors through the selected vertex

**for (int v = 0; v < g->vertices; ++v) {**

**if (!visited[v] && g->graph[u][v] && dist[u] != INFINITY && dist[u] + g->graph[u][v] < dist[v]) {**

**dist[v] = dist[u] + g->graph[u][v];** // Update the distance

**pred[v] = u;**  // Update the predecessor

**}**

**}**

**}**

// Print the shortest distances from the source vertex

**printf("Vertex \t Distance from Source\n");**

**for (int i = 0; i < g->vertices; ++i) {**

**printf("%d \t\t %d\n", i, dist[i]);**

**}**

**}**

**int main() {**

**scanf("%d", &vertices);**

// Allocate memory for the graph structure

**Graph\* g = malloc(sizeof(Graph));**

**g->vertices = vertices;**

// Input the adjacency matrix representing the graph

**printf("Enter the adjacency matrix:\n");**

**for (int i = 0; i < vertices; ++i) {**

**for (int j = 0; j < vertices; ++j) {**

**scanf("%d", &g->graph[i][j]);**

**}**

**}**

**int start;**

**printf("Enter the source vertex: ");**

**scanf("%d", &start);**

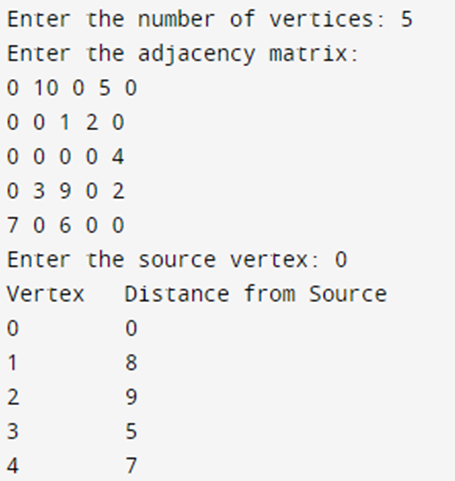
**dijkstra(g, start);**

// Free dynamically allocated memory

**free(g);**

**return 0;**

**}**



**Worst Case:** **O()**.

**Normally: O(V+E)**.