selection sort

```c
#include <stdio.h>


// Function to swap two elements

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

    int temp = *a;

    *a = *b;

    *b = temp;

}


// Function to perform selection sort

void selectionSort(int arr[], int n) {

    int i, j, min_idx;


    // One by one move boundary of unsorted subarray

    for (i = 0; i < n-1; i++) {

        // Find the minimum element in unsorted array

        min_idx = i;

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

            if (arr[j] < arr[min_idx])

                min_idx = j;

        }


        // Swap the found minimum element with the first element

        swap(&arr[min_idx], &arr[i]);

    }

}
```

```
// Function to print an array
void printArray(int arr[], int size) {
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");
}
int main() {
   int arr[] = {64, 25, 12, 22, 11};
   int n = sizeof(arr)/sizeof(arr[0]);
   printf("Unsorted array: \n");
   printArray(arr, n);

   selectionSort(arr, n);

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

insertion sort

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
```

```
        // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their
current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(int arr[], int n) {
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: \n");
    printArray(arr, n);
    insertionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

merge sort

```
#include <stdio.h>

// Merge two sorted subarrays arr[l..m] and arr[m+1..r]

void merge(int arr[], int l, int m, int r) {

  int i, j, k;

  int n1 = m - l + 1;

  int n2 = r - m;

  // Create temporary arrays

  int L[n1], R[n2];

  // Copy data to temporary arrays L[] and R[]

  for (i = 0; i < n1; i++)

    L[i] = arr[l + i];

  for (j = 0; j < n2; j++)

    R[j] = arr[m + 1 + j];

  // Merge the temporary arrays back into arr[l..r]

  i = 0;

  j = 0;

  k = l;

  while (i < n1 && j < n2) {

    if (L[i] <= R[j]) {

      arr[k] = L[i];

      i++;

    } else {

      arr[k] = R[j];

      j++;

    }

    k++;   }
```

```
    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
// Main function that sorts arr[l..r] using merge()
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for large l and r
        int m = l + (r - l) / 2;

       // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

```
// Function to print an array
void printArray(int A[], int size) {
   int i;
   for (i = 0; i < size; i++)
      printf("%d ", A[i]);
   printf("\n");
}
int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int arr_size = sizeof(arr) / sizeof(arr[0]);
   printf("Given array is \n");
   printArray(arr, arr_size);
  mergeSort(arr, 0, arr_size - 1);
   printf("\nSorted array is \n");
   printArray(arr, arr_size);
   return 0;
}
```

Quick sort

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
```

```
// Function to partition the array using the last element as pivot
int partition(int arr[], int low, int high) {
   int pivot = arr[high];    // pivot
   int i = (low - 1);  // Index of smaller element
   for (int j = low; j <= high - 1; j++) {
      // If current element is smaller than or equal to pivot
      if (arr[j] <= pivot) {
         i++;    // increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
// Function to implement quicksort
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      // pi is partitioning index, arr[pi] is now at right place
      int pi = partition(arr, low, high);

      // Separately sort elements before and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
// Function to print an array
void printArray(int arr[], int size) {
```

```
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
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;
}
```

Finding minimum and maximum element

```
#include <stdio.h>
// Function to find the minimum and maximum elements in an array
void findMinMax(int arr[], int n) {
    int min = arr[0];
    int max = arr[0];
    // Traverse the array to find minimum and maximum elements
    for (int i = 1; i < n; i++) {
        if (arr[i] < min) {
            min = arr[i];
        } else if (arr[i] > max) {
            max = arr[i];
```

```
        }
    }
    // Print the minimum and maximum elements
    printf("Minimum element: %d\n", min);
    printf("Maximum element: %d\n", max);
}
int main() {
    int arr[] = {3, 8, 1, 5, 10, 4};
    int n = sizeof(arr) / sizeof(arr[0]);
    // Call the function to find minimum and maximum elements
    findMinMax(arr, n);
    return 0;
}
```

(Fractional knapsack Problem)

```
#include <stdio.h>
#include <stdlib.h>
// Structure to represent each item
struct Item {
    int value;
    int weight;
};
// Comparison function to sort items based on value per unit weight
int compare(const void *a, const void *b) {
    double ratio1 = (double)((struct Item *)a)->value / ((struct Item *)a)->weight;
    double ratio2 = (double)((struct Item *)b)->value / ((struct Item *)b)->weight;
    if (ratio1 < ratio2)
        return 1; // Sort in descending order
    else if (ratio1 > ratio2)
```

```
        return -1;
    else
        return 0;
}
// Function to solve the Fractional Knapsack Problem
void fractionalKnapsack(struct Item items[], int n, int capacity) {
    // Sort items based on value per unit weight
    qsort(items, n, sizeof(struct Item), compare);
    double totalValue = 0.0;
    int currentWeight = 0;
    // Iterate through sorted items and add them to the knapsack
    for (int i = 0; i < n; i++) {
        // If adding the entire item is possible
        if (currentWeight + items[i].weight <= capacity) {
            currentWeight += items[i].weight;
            totalValue += items[i].value;
        } else { // If adding only a fraction of the item is possible
            int remainingWeight = capacity - currentWeight;
            totalValue += (double)items[i].value * remainingWeight / items[i].weight;
            break;
        }
    }
    // Print the total value of items in the knapsack
    printf("Total value in knapsack: %.2lf\n", totalValue);
}
int main() {
    // Example items
```

```
    struct Item items[] = {{60, 10}, {100, 20}, {120, 30}};
    int n = sizeof(items) / sizeof(items[0]);
    int capacity = 50;
    // Solve the Fractional Knapsack Problem
    fractionalKnapsack(items, n, capacity);
    return 0;
}
```

All pair Shortest Path: Floyd Warshall Algorithm

```
#include <stdio.h>
#include <limits.h>
#define V 4 // Number of vertices in the graph
// Function to print the solution matrix
void printSolution(int dist[][V]) {
    printf("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\t");
            else
                printf("%d\t", dist[i][j]);
        }
        printf("\n");
    }
}

// Implementation of Floyd Warshall algorithm
void floydWarshall(int graph[][V]) {
```

```
    int dist[V][V];

    // Initialize the solution matrix same as input graph matrix
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

   // Add all vertices one by one to the set of intermediate vertices
    for (int k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (int i = 0; i < V; i++) {
            // Pick all vertices as destination for the above picked source
            for (int j = 0; j < V; j++) {
                // If vertex k is on the shortest path from i to j,
                // then update the value of dist[i][j]
                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];
            }
        }
    }

    printSolution(dist);
}

int main() {
    // Example graph represented as an adjacency matrix
    int graph[V][V] = {
        {0, 5, INT_MAX, 10},
```

```
        {INT_MAX, 0, 3, INT_MAX},
        {INT_MAX, INT_MAX, 0, 1},
        {INT_MAX, INT_MAX, INT_MAX, 0}
    };
 // Call the Floyd Warshall algorithm function
    floydWarshall(graph);

    return 0;
}
```

Longest Common Subsequence

```
#include <stdio.h>
#include <string.h>
// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}
// Function to find the length of the longest common subsequence
int lcs(char *X, char *Y, int m, int n) {
    int L[m + 1][n + 1];
    int i, j;
    // Build L[m+1][n+1] in bottom-up fashion
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
```

```c
            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    // L[m][n] contains the length of LCS of X[0..m-1] and Y[0..n-1]
    return L[m][n];
}
int main() {
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    int m = strlen(X);
    int n = strlen(Y);
    printf("Length of Longest Common Subsequence: %d\n", lcs(X, Y, m, n));
    return 0;
}
```