TW 1: LINEAR SEARCH

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

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

int main() {
    int arr[10000], n, key, i, index;
    clock_t s, e;
    float t;

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

    for (i = 0; i < n; i++)
        arr[i] = rand();

    printf("Array elements are: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    printf("\nEnter the key to search: ");
    scanf("%d", &key);

    s = clock();
    for (i = 0; i < 1000; i++)
        index = linearSearch(arr, n, key);
    e = clock();

    if (index != -1)
        printf("%d found at index %d\n", key, index);
    else
        printf("%d not found\n", key);

    t = (float)(e - s) / CLOCKS_PER_SEC;
    printf("\nExecution time is %f ", t);

    return 0;
}
```

OUTPUT:

```
Enter number of elements: 400
4895…
Enter the key to search: 8177
8177 found at index 399

Execution time is 0.000000
```

TW2: BINARY SEARCH

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

int binary_search(int arr[], int size, int key) {
   int low = 0, high = size - 1, mid;
   while (low <= high) {
      mid = (low + high) / 2;
      if (arr[mid] == key)
         return 1;
      if (key < arr[mid])
         high = mid - 1;
      else
         low = mid + 1;
   }
   return 0;
}

int main() {
   int n, key, found = 0;
   clock_t start, end;
   float time_taken;
   const int repetitions = 10000;

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

   int arr[n];

   // Generate sorted array
   for (int i = 0; i < n; i++)
      arr[i] = i;

   // Print array elements
   printf("\nArray elements:\n");
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);

   printf("\n\nEnter key to search: ");
   scanf("%d", &key);

   // Measure execution time for 10000 runs
   start = clock();
   for (int i = 0; i < repetitions; i++) {
      found = binary_search(arr, n, key);
   }
   end = clock();

   // Output result (check one final time)
   if (binary_search(arr, n, key))
      printf("\nKey FOUND in the array.\n");
   else
      printf("\nKey NOT FOUND in the array.\n");
```

```
    // Print average execution time
    time_taken = (float)(end - start) / CLOCKS_PER_SEC;
    printf("Total time taken by the algorithm (for %d searches): %f seconds\n", repetitions, time_taken);
    printf("Average time per search: %f seconds\n", time_taken / repetitions);

    return 0;
}
```

Output:

```
Enter number of elements: 10000
4566..
Enter key to search: 9977

Key FOUND in the array.
Total time taken by the algorithm (for 10000 searches): 0.001000 seconds
```

TW 3 : MERGE SORT

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

void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;

   // Allocate temporary arrays dynamically
   int *L = (int *)malloc(n1 * sizeof(int));
   int *R = (int *)malloc(n2 * sizeof(int));

   for (i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

   i = 0;
   j = 0;
   k = left;

   while (i < n1 && j < n2) {
      if (L[i] <= R[j])
         arr[k++] = L[i++];
      else
         arr[k++] = R[j++];
   }

   while (i < n1)
      arr[k++] = L[i++];
   while (j < n2)
      arr[k++] = R[j++];

   free(L);
   free(R);
}

void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = (left + right) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }
}

int main() {
   int n, i;
   clock_t s, e;
   float t;
```

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

    int *arr = (int *)malloc(n * sizeof(int));
    srand(time(0)); // Seed the random number generator

    for (i = 0; i < n; i++)
        arr[i] = rand();

    printf("\nUnsorted Array:\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    s = clock();
    mergeSort(arr, 0, n - 1);
    e = clock();

    printf("\n\nSorted Array:\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);

    t = (float)(e - s) / CLOCKS_PER_SEC;
    printf("\n\nTime Taken to sort is %f seconds\n", t);

    free(arr);
    return 0;
}
```

Output:

```
Enter number of elements: 300

Unsorted Array:5678…
Sorted Array: …
Time Taken to sort is 0.000000 seconds
```

TW 4 : BFS

```
#include <stdio.h>
#define SIZE 20

int queue[SIZE], front = 0, rear = -1;
int visited[SIZE];

void bfs(int adj[SIZE][SIZE], int n, int start) {
    int i;
    visited[start] = 1;
    queue[++rear] = start;

    while (front <= rear) {
        int node = queue[front++];
        printf("%d ", node);
        for (i = 0; i < n; i++) {
            if (adj[node][i] && !visited[i]) {
                queue[++rear] = i;
                visited[i] = 1;
            }}}}

int main() {
    int n, adj[SIZE][SIZE], i, j, start;

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

    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &adj[i][j]);

    printf("Enter starting vertex: ");
    scanf("%d", &start);

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

    printf("BFS traversal: ");
    bfs(adj, n, start);
    return 0;
}
```

Output:

```
Enter number of vertices: 4
Enter adjacency matrix:
2 3 0 0
2 4 1 0
2 9 4 0
0 3 0 1
Enter starting vertex: 1
BFS traversal: 1 0 2
```

Tw 5 : prim's algorithm

```
#include <stdio.h>

int a, b, u, v, n, i, j, ne = 1;
int visited[10] = {0}, min, mincost = 0, cost[10][10];

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

   printf("Enter the adjacency matrix:\n");
   for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
         scanf("%d", &cost[i][j]);
         if (cost[i][j] == 0)
            cost[i][j] = 999; // Represent no edge with high cost
      }
   }

   visited[1] = 1;
   printf("\n");

   while (ne < n) {
      min = 999;
      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            if (cost[i][j] < min) {
               if (visited[i] != 0 && visited[j] == 0) {
                  min = cost[i][j];
                  a = u = i;
                  b = v = j;
               }
            }
         }
      }

      if (visited[u] == 0 || visited[v] == 0) {
         printf("Edge %d: (%d %d) cost: %d\n", ne++, a, b, min);
         mincost += min;
         visited[b] = 1;
      }

      cost[a][b] = cost[b][a] = 999; // Mark edge as used
   }

   printf("Minimum cost = %d\n", mincost);

   return 0;
}
```

Output :

```
Enter the number of nodes: 4
Enter the adjacency matrix:
```

```
0 4 0 6
4 0 5 0
0 5 0 7
6 0 7 0

Edge 1: (1 2) cost: 4
Edge 2: (2 3) cost: 5
Edge 3: (1 4) cost: 6
Minimum cost = 15
```

Tw 6: dikstra’s algorithm

```
#include<stdio.h>
#define INFINITY 999
#define MAX 50

void dijkstra(int G[MAX][MAX], int n, int startnode);

int main() {
   int G[MAX][MAX], i, j, n, u, flag = 0;

   printf("Graph: Shortest Path to Other Vertices: Dijkstra Algorithm >>\n\n");
   printf("Enter Number of Vertices Present in the Graph: ");
   scanf("%d", &n);

   printf("\nEnter the Adjacency Matrix:\n");
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         scanf("%d", &G[i][j]);

   printf("\nEnter The Starting Node: ");
   scanf("%d", &u);

   printf("\nSo, The Adjacency Matrix is:\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         printf("%d ", G[i][j]);
      }
      printf("\n");
   }

   dijkstra(G, n, u);

   do {
      printf("\nWant to Continueů. 1 for Yes, 0 for No : ");
      scanf("%d", &flag);
      if (flag == 1) {
         printf("\n\nEnter The Starting Node: ");
         scanf("%d", &u);
         dijkstra(G, n, u);
      }
   } while (flag == 1);

   return 0;
}

void dijkstra(int G[MAX][MAX], int n, int startnode) {
   int cost[MAX][MAX], distance[MAX], pred[MAX];
   int visited[MAX], count, mindistance, nextnode, i, j;

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         if (G[i][j] == 0)
            cost[i][j] = INFINITY;
            else
                cost[i][j] = G[i][j];

    for (i = 0; i < n; i++) {
        distance[i] = cost[startnode][i];
        pred[i] = startnode;
        visited[i] = 0;
    }

    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;

    while (count < n - 1) {
        mindistance = INFINITY;
        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }

        visited[nextnode] = 1;
        for (i = 0; i < n; i++)
            if (!visited[i])
                if (mindistance + cost[nextnode][i] < distance[i]) {
                    distance[i] = mindistance + cost[nextnode][i];
                    pred[i] = nextnode;
                }
        count++;
    }

    for (i = 0; i < n; i++) {
        if (i != startnode) {
            if (distance[i] == 999) {
                printf("\nThere is no Possible Path Between %d and %d.", i, startnode);
            } else {
                printf("\nDistance of Node %d to %d is: %d", i, startnode, distance[i]);
                printf("\nAnd the Path is: %d ", i);
                j = i;
                do {
                    j = pred[j];
                    printf(" -> %d", j);
                } while (j != startnode);
            }
        }
        printf("\n");
    }
}
```

Output :

```
Graph: Shortest Path to Other Vertices: Dijkstra Algorithm >>

Enter Number of Vertices Present in the Graph: 5
Enter the Adjacency Matrix:
2 3 4 6 7
3 4 2 6 7
1 4 7 8 3
2 4 6 8 4
2 6 8 5 8

Enter The Starting Node: 3

So, The Adjacency Matrix is:
2 3 4 6 7
3 4 2 6 7
1 4 7 8 3
2 4 6 8 4
2 6 8 5 8

Distance of Node 0 to 3 is: 2
And the Path is: 0  -> 3

Distance of Node 1 to 3 is: 4
And the Path is: 1  -> 3

Distance of Node 2 to 3 is: 6
And the Path is: 2  -> 3


Distance of Node 4 to 3 is: 4
And the Path is: 4  -> 3

Want to Continue┼». 1 for Yes, 0 for No : 1
```

Tw 7: 0/1 knapsack problem

```
#include <stdio.h>
#define MAX 200

int V[MAX][MAX] = {0};
int res[200] = {0};
int count = 0;

int max(int a, int b) {
  return (a > b) ? a : b;
}

int knapSack(int W, int wt[], int val[], int n) {
  int i, j;
  for (i = 0; i <= n; i++) {
    for (j = 0; j <= W; j++) {
      if (i == 0 || j == 0) {
        V[i][j] = 0;
      } else if (wt[i - 1] <= j) {
        V[i][j] = max(val[i - 1] + V[i - 1][j - wt[i - 1]], V[i - 1][j]);
      } else {
        V[i][j] = V[i - 1][j];
      }
    }

    // Print matrix after each item iteration
    printf("DP Table after considering item %d:\n", i);
    for (int k = 0; k <= n; k++) {
      for (int m = 0; m <= W; m++) {
        printf("%d ", V[k][m]);
      }
      printf("\n");
    }
    printf("\n");
  }

  // Traceback to find selected items
  i = n;
  j = W;
  while (i > 0 && j > 0) {
    if (V[i][j] != V[i - 1][j]) {
      res[count++] = i;
      j = j - wt[i - 1];
    }
    i--;
  }

  return V[n][W];
}

int main() {
  int i, n, W, optsoln;
  int val[20], wt[20];
```

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

    printf("Enter the weights:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &wt[i]);

    printf("Enter the values:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &val[i]);

    printf("Enter the knapsack capacity: ");
    scanf("%d", &W);

    optsoln = knapSack(W, wt, val, n);

    printf("\nThe optimal solution is: %d", optsoln);
    printf("\nItems included in knapsack are:");
    for (i = count - 1; i >= 0; i--)
        printf(" %d", res[i]);

    printf("\n");
    return 0;
}
```

Output :

```
Enter number of items:
4
Enter the weights:
2 1 3 2
Enter the values:
12 10 20 15
Enter the knapsack capacity: 5

DP Table after considering item 0:
...

The optimal solution is: 37
Items included in knapsack are: 4 2
```

Tw 8: floyed’s algorithm

```
#include <stdio.h>
#define MAX 10
#define INF 999

void floyd(int w[MAX][MAX], int n) {
   int i, j, k;

   for (k = 1; k <= n; k++) {
      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            if (w[i][k] + w[k][j] < w[i][j]) {
               w[i][j] = w[i][k] + w[k][j];
            }
         }
      }

      // Print matrix D[k]
      printf("\nMatrix D[%d]:\n", k);
      for (i = 1; i <= n; i++) {
         for (j = 1; j <= n; j++) {
            if (w[i][j] == INF)
               printf("%4s", "INF");
            else
               printf("%4d", w[i][j]);
         }
         printf("\n");
      }
   }
}

int main() {
   int w[MAX][MAX], i, j, n;

   printf("Floyd's Algorithm - All Pairs Shortest Path\n");
   printf("Enter the number of nodes: ");
   scanf("%d", &n);

   printf("Enter the adjacency matrix (use 999 for no direct edge):\n");
   for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
         scanf("%d", &w[i][j]);
      }
   }

   floyd(w, n);

   return 0;
}
```

Output

```
Enter the number of nodes: 4
Enter the adjacency matrix (use 999 for no direct edge):
```

```
0 5 999 10
999 0 3 999
999 999 0 1
999 999 999 0

Matrix D[1]:
  0   5 INF  10
INF   0   3 INF
INF INF   0   1
INF INF INF   0

Matrix D[2]:
  0   5   8  10
INF   0   3 INF
INF INF   0   1
INF INF INF   0
...
```

Tw 9: N-queen's problem

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

int a[30], count = 0;  // 'a[i]' stores column position of queen in row 'i'

int place(int pos) {
   int i;
   for (i = 1; i < pos; i++) {
      // Check if same column or same diagonal
      if ((a[i] == a[pos]) || (abs(a[i] - a[pos]) == abs(i - pos)))
         return 0;
   }
   return 1;
}

void printsol(int n) {
   int i, j;
   count++;
   printf("\n\nSolution #%d:\n\n", count);
   for (i = 1; i <= n; i++) {
      for (j = 1; j <= n; j++) {
         if (a[i] == j)
            printf("Q\t");
         else
            printf("*\t");
      }
      printf("\n");
   }
}

void queen(int n) {
   int k = 1;
   a[k] = 0;
```

```
    while (k != 0) {
        a[k] += 1;  // Try next column in row k

        while (a[k] <= n && !place(k))  // Check until a valid column is found or end is reached
            a[k]++;

        if (a[k] <= n) {
            if (k == n)
                printsol(n);  // Found a solution
            else {
                k++;
                a[k] = 0;     // Start from 0 for the next row
            }
        } else {
            k--;  // Backtrack
        }
    }
}

int main() {
    int n;
    printf("Enter the number of queens: ");
    scanf("%d", &n);
    queen(n);
    printf("\nTotal Number of Solutions = %d\n", count);
    return 0;
}
```

Output :

```
Enter the number of queens: 5


Solution #1:

Q    *    *    *    *
*    *    Q    *    *
*    *    *    *    Q
*    Q    *    *    *
*    *    *    Q    *


Solution #2:

Q    *    *    *    *
*    *    *    Q    *
*    Q    *    *    *
*    *    *    *    Q
*    *    Q    *    *


Solution #3:

*    Q    *    *    *
*    *    *    Q    *
Q    *    *    *    *
*    *    Q    *    *
*    *    *    *    Q
```

Tw 10: tsp problem

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

int a[10][10], visited[10], n;
int minCost = INT_MAX;
int path[10], tempPath[10];

void get() {
    int i, j;

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

    printf("Enter Cost Matrix:\n");
    for (i = 0; i < n; i++) {
        printf("Enter elements of row %d:\n", i + 1);
        for (j = 0; j < n; j++) {
            scanf("%d", &a[i][j]);
        }
        visited[i] = 0;
    }

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

void tsp(int currPos, int count, int cost, int start) {
    visited[currPos] = 1;
    tempPath[count - 1] = currPos;

    if (count == n && a[currPos][start]) {
        int totalCost = cost + a[currPos][start];
        if (totalCost < minCost) {
            minCost = totalCost;
            for (int i = 0; i < n; i++)
                path[i] = tempPath[i];
        }
        visited[currPos] = 0;
        return;
    }
    for (int i = 0; i < n; i++) {
        if (!visited[i] && a[currPos][i]) {
            tsp(i, count + 1, cost + a[currPos][i], start);
        }
    }

    visited[currPos] = 0;
}

int main() {
    get();
    tsp(0, 1, 0, 0);

    printf("\n\nThe Minimum Cost Path is:\n");
    for (int i = 0; i < n; i++) {
        printf("%d -> ", path[i] + 1);
    }
    printf("1");

    printf("\n\nMinimum cost: %d\n", minCost);
    return 0;
}
```

Output:

```
Enter number of cities: 5
Enter Cost Matrix:
Enter elements of row 1:
1 2 3 4 5
Enter elements of row 2:
2 3 4 5 6
Enter elements of row 3:
2 3 4 5 6
Enter elements of row 4:
5 6 7 8 6
Enter elements of row 5:
6 7 8 9 9

The cost matrix is:
        1       2       3       4       5
        2       3       4       5       6
        2       3       4       5       6
        5       6       7       8       6
        6       7       8       9       9


The Minimum Cost Path is:
1 -> 2 -> 3 -> 4 -> 5 -> 1

Minimum cost: 23
```