Quick sort (Exp 1):

```
#include<stdio.h>
void quicksort(int num[25], int first, int last)
{
	int i, j, pivot, temp;
	if(first<last)
	{
		pivot = first;
		i= first;
		j= last;
		while(i<j)
		{
			while(num[i]<=num[pivot]&&i<last)
			i++;
			while(num[j]>num[pivot])
			j--;
			if(i<j)
			{
				temp = num[i];
				num[i] = num[j];
				num[j] = temp;
			}
		}
		temp = num[pivot];
		num[pivot] = num[j];
		num[j] = temp;
		quicksort(num,first,j-1);
	   quicksort(num,j+1,last);
	}
}
int main()
{
    int i, count, num[25];
    printf("How many elements are u going to enter?: ");
    scanf("%d",&count);
    printf("Enter %d elements: ", count);
    for(i=0;i<count;i++)
    scanf("%d",&num[i]);
    quicksort(num,0,count-1);
    printf("Order of Sorted elements: ");
    for(i=0;i<count;i++)
    printf(" %d",num[i]);
    return 0;
}
```

Merge Sort (Exp 2):

```
#include <stdio.h>
void merge(int a[], int beg, int mid, int end)
{
	int i, j, k;
	int n1 = mid - beg + 1;
	int n2 = end - mid;

	int LeftArray[n1], RightArray[n2];

	for (int i = 0; i < n1; i++)
	LeftArray[i] = a[beg + i];
	for (int j = 0; j < n2; j++)
	RightArray[j] = a[mid + 1 + j];

	i = 0;
	j = 0;
	k = beg;

	while (i < n1 && j < n2)
	{
		if(LeftArray[i] <= RightArray[j])
		{
			a[k] = LeftArray[i];
			i++;
		}
		else
		{
			a[k] = RightArray[j];
			j++;
		}
		k++;
	}
	while (i<n1)
	{
		a[k] = LeftArray[i];
		i++;
		k++;
	}
	while (j<n2)
	{
		a[k] = RightArray[j];
		j++;
		k++;
	}
}
```

```
void mergeSort(int a[], int beg, int end)
{
	if (beg < end)
	{
		int mid = (beg + end) / 2;
		mergeSort(a, beg, mid);
		mergeSort(a, mid + 1, end);
		merge(a, beg, mid, end);
	}
}
void printArray(int a[], int n)
{
	int i;
	for (i = 0; i < n; i++)
		printf("%d ", a[i]);
	printf("¥n");
}
int main()
{
	int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
	int n = sizeof(a) / sizeof(a[0]);
	printf("Before sorting array elements are - ¥n");
	printArray(a, n);
	mergeSort(a, 0, n - 1);
	printf("After sorting array elements are - ¥n");
	printArray(a, n);
	return 0;
}
```

Knapsack Problem (Exp 3):

```
# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity) {
    float x[20], tp = 0;
    int i, j, u;
    u = capacity;

    for (i = 0; i < n; i++)
        x[i] = 0.0;

    for (i = 0; i < n; i++) {
        if (weight[i] > u)
            break;
        else {
```

```
            x[i] = 1.0;
            tp = tp + profit[i];
            u = u - weight[i];
        }
    }

    if (i < n)
        x[i] = u / weight[i];

    tp = tp + (x[i] * profit[i]);
    printf("¥nThe result vector is:- ");
    for (i = 0; i < n; i++)
        printf("%f¥t", x[i]);

    printf("¥nMaximum profit is:- %f", tp);
}

int main() {
    float weight[20], profit[20], capacity;
    int num, i, j;
    float ratio[20], temp;

    printf("¥nEnter the no. of objects:- ");
    scanf("%d", &num);

    printf("¥nEnter the wts and profits of each object:- ");
    for (i = 0; i < num; i++) {
        scanf("%f %f", &weight[i], &profit[i]);
    }

    printf("¥nEnter the capacityacity of knapsack:- ");
    scanf("%f", &capacity);

    for (i = 0; i < num; i++) {
        ratio[i] = profit[i] / weight[i];
    }
    for (i = 0; i < num; i++) {
        for (j = i + 1; j < num; j++) {
            if (ratio[i] < ratio[j]) {
                temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;
```

```
                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }

    knapsack(num, weight, profit, capacity);
    return(0);
}
```

Exp 4:

a) Kruskal's Algorithm:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
        clrscr();
        printf("¥n¥tImplementation of Kruskal's algorithm¥n");
        printf("¥nEnter the no. of vertices:");
        scanf("%d",&n);
        printf("¥nEnter the cost 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;
                }
        }
        printf("The edges of Minimum Cost Spanning Tree are¥n");
        while(ne < n)
        {
                for(i=1,min=999;i<=n;i++)
                {
                        for(j=1;j <= n;j++)
                        {
                                if(cost[i][j] < min)
```

```
                                        {
                                                min=cost[i][j];
                                                a=u=i;
                                                b=v=j;
                                        }
                                }
                        }
                        u=find(u);
                        v=find(v);
                        if(uni(u,v))
                        {
                                printf("%d edge (%d,%d) =%d¥n",ne++,a,b,min);
                                mincost +=min;
                        }
                        cost[a][b]=cost[b][a]=999;
                }
                printf("¥n¥tMinimum cost = %d¥n",mincost);
                getch();
        }
        int find(int i)
        {
                while(parent[i])
                i=parent[i];
                return i;
        }
        int uni(int i,int j)
        {
                if(i!=j)
                {
                        parent[j]=i;
                        return 1;
                }
                return 0;
        }
```

b) Prims Algorithm:

```
#include <stdio.h>
#include <limits.h>
#define V 5

int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, min_index;
    int v;
    for (v = 0; v < V; v++)
```

```
            if (mstSet[v] == 0 && key[v] < min)
                min = key[v], min_index = v;

    return min_index;
}

int printMST(int parent[], int n, int graph[V][V]) {
    int i;
    printf("Edge      Weight¥n");
    for (i = 1; i < V; i++)
        printf("%d - %d        %d ¥n", parent[i], i, graph[i][parent[i]]);
        return 0;
}

void primMST(int graph[V][V]) {
    int parent[V]; // Array to store constructed MST
    int key[V], i, v, count; // Key values used to pick minimum weight edge in cut
    int mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;

        for (v = 0; v < V; v++)

            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, V, graph);
}

int main() {
    /* Let us create the following graph
      2       3
    (0)--(1)--(2)
      |     / ¥     |
```

```
        6| 8/      ¥5 |7
          | /         ¥ |
        (3)-------(4)
          9                   */
        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);

        return 0;
}
```

Floyd Warshall Algorithm (Exp 5):

```
#include <stdio.h>
#define V 4
#define INF 99999

void printSolution(int dist[][V]);

void floydWarshall(int dist[][V])
{
        int i, j, k;

        for (k = 0; k < V; k++) {

                for (i = 0; i < V; i++) {

                        for (j = 0; j < V; j++) {

                                if (dist[i][k] + dist[k][j] < dist[i][j])
                                        dist[i][j] = dist[i][k] + dist[k][j];
                        }
                }
        }

        printSolution(dist);
}

void printSolution(int dist[][V])
{
        printf(
                "The following matrix shows the 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] == INF)
                    printf("%7s", "INF");
                else
                    printf("%7d", dist[i][j]);
            }
            printf("¥n");
        }
}

int main()
{
        /* Let us create the following weighted graph
                    10
             (0)------->(3)
               |               /|¥
           5 |                |
               |                | 1
             ¥|/               |
             (1)------->(2)
                    3                   */
        int graph[V][V] = { { 0, 5, INF, 10 },
                                        { INF, 0, 3, INF },
                                        { INF, INF, 0, 1 },
                                        { INF, INF, INF, 0 } };
        // Function call
        floydWarshall(graph);
        return 0;
}
```

Longest Common Subsequence (Exp 6):

```
#include <stdio.h>
#include <string.h>
int i, j, m, n, LCS_table[20][20];
char S1[20] = "abaaba", S2[20] = "babbab", b[20][20];
void lcsAlgo() {
  m = strlen(S1);
  n = strlen(S2);
 // Filling 0's in the matrix
  for (i = 0; i <= m; i++)
    LCS_table[i][0] = 0;
  for (i = 0; i <= n; i++)
    LCS_table[0][i] = 0;
 // Creating the mtrix in bottom-up way
  for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
```

```
    if (S1[i - 1] == S2[j - 1]) {
      LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
    } else if (LCS_table[i - 1][j] >= LCS_table[i][j - 1]) {
      LCS_table[i][j] = LCS_table[i - 1][j];
    } else {
      LCS_table[i][j] = LCS_table[i][j - 1];
    }
   }

  int index = LCS_table[m][n];
  char lcsAlgo[index + 1];
  lcsAlgo[index] = '¥0';

  int i = m, j = n;
  while (i > 0 && j > 0) {
   if (S1[i - 1] == S2[j - 1]) {
    lcsAlgo[index - 1] = S1[i - 1];
    i--;
    j--;
    index--;
   }

   else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
    i--;
   else
    j--;
  }

  // Printing the sub sequences
  printf("S1 : %s ¥nS2 : %s ¥n", S1, S2);
  printf("LCS: %s", lcsAlgo);
}

int main() {
  lcsAlgo();
  printf("¥n");
}
```

N Queens Using Backtracking (Exp 7):

```
#define N 4
#include <stdbool.h>
#include <stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])
{
```

```
	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++)
			printf(" %d ", board[i][j]);
		printf("¥n");
	}
}

bool isSafe(int board[N][N], int row, int col)
{
	int i, j;
	for (i = 0; i < col; i++)
		if (board[row][i])
			return false;
	for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
		if (board[i][j])
			return false;

	for (i = row, j = col; j >= 0 && i < N; i++, j--)
		if (board[i][j])
			return false;

	return true;
}

bool solveNQUtil(int board[N][N], int col)
{

	if (col >= N)
		return true;

	for (int i = 0; i < N; i++) {

		if (isSafe(board, i, col)) {

			board[i][col] = 1;

			if (solveNQUtil(board, col + 1))
				return true;

			board[i][col] = 0; // BACKTRACK
		}
	}

	return false;
}

bool solveNQ()
```

```
{
	int board[N][N] = { { 0, 0, 0, 0 },
					{ 0, 0, 0, 0 },
					{ 0, 0, 0, 0 },
					{ 0, 0, 0, 0 } };

	if (solveNQUtil(board, 0) == false) {
		printf("Solution does not exist");
		return false;
	}

	printSolution(board);
	return true;
}

int main()
{
	solveNQ();
	return 0;
}
```

Graph Colouring using Backtracking (Exp 8):

```
#include<iostream>
#define NODE 6
using namespace std;

int graph[NODE][NODE] = {
   {0, 1, 1, 1, 0, 0},
   {1, 0, 0, 1, 1, 0},
   {1, 0, 0, 1, 0, 1},
   {1, 1, 1, 0, 1, 1},
   {0, 1, 0, 1, 0, 1},
   {0, 0, 1, 1, 1, 0}
};

void graphColoring() {
   int color[NODE];
   color[0] = 0;       //Assign first color for the first node
   bool colorUsed[NODE];       //Used to check whether color is used or not

   for(int i = 1; i<NODE; i++)
      color[i] = -1;       //initialize all other vertices are unassigned

   for(int i = 0; i<NODE; i++)
      colorUsed[i] = false;       //initially any colors are not chosen
```

```
    for(int u = 1; u<NODE; u++) {        //for all other NODE - 1 vertices
        for(int v = 0; v<NODE; v++) {
            if(graph[u][v]){
                if(color[v] != -1)        //when one color is assigned, make it unavailable
                    colorUsed[color[v]] = true;
            }
        }

        int col;
        for(col = 0; col<NODE; col++)
            if(!colorUsed[col])        //find a color which is not assigned
                break;

        color[u] = col;        //assign found color in the list

        for(int v = 0; v<NODE; v++) {        //for next iteration make color availability to false
            if(graph[u][v]) {
                if(color[v] != -1)
                    colorUsed[color[v]] = false;
            }
        }
    }

    for(int u = 0; u<NODE; u++)
        cout <<"Color: " << u << ", Assigned with Color: " <<color[u] <<endl;
}

main() {
    graphColoring();
}
```

Naive String Matching Algorithm (Exp 9):

Old:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int match(char st[100], char pat[100]);
int main(int argc, char **argv) {
      char st[100], pat[100];
      int status;
      printf("*** Naive String Matching Algorithm ***¥n");
      printf("Enter the String.¥n");
      gets(st);
```

```
	printf("Enter the pattern to match.¥n");
	gets(pat);
	status = match(st, pat);
	if (status == -1)
		printf("¥nNo match found");
	else
		printf("Match has been found on %d position.", status);
	return 0;
}
int match(char st[100], char pat[100]) {
	int n, m, i, j, count = 0, temp = 0;
	n = strlen(st);
	m = strlen(pat);
	for (i = 0; i <= n - m; i++) {
		temp++;
		for (j = 0; j < m; j++) {
			if (st[i + j] == pat[j])
				count++;
		}
		if (count == m)
			return temp;
		count = 0;
	}
	return -1;
}
```

New:

```
#include <stdio.h>
#include <string.h>
void search(char* pat, char* txt)
{
int M = strlen(pat);
int N = strlen(txt);
for (int i = 0; i <= N - M; i++)
{
int j;
for (j = 0; j < M; j++)
if (txt[i + j] != pat[j])
break;
if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
printf("Pattern found at index %d ¥n", i);
}
}
int main()
{
```

```
char txt[50];
char pat[50];
printf("**Naive String Matching Algorithm **¥n");
printf("Enter a string:");
gets(txt);
printf("Enter the pattern:");
gets(pat);
search(pat, txt);
return 0;
}
```

Rabin Karp Algorithm (Exp 10):

```
#include <stdio.h>
#include <string.h>
#define d 256

void search(char pat[], char txt[], int q)
{
	int M = strlen(pat);
	int N = strlen(txt);
	int i, j;
	int p = 0;
	int t = 0;
	int h = 1;

	for (i = 0; i < M - 1; i++)
		h = (h * d) % q;

	for (i = 0; i < M; i++) {
		p = (d * p + pat[i]) % q;
		t = (d * t + txt[i]) % q;
	}

	for (i = 0; i <= N - M; i++) {

		if (p == t) {

			for (j = 0; j < M; j++) {
				if (txt[i + j] != pat[j])
					break;
			}

			if (j == M)
				printf("Pattern found at index %d ¥n", i);
```

```
            }

            if (i < N - M) {
                t = (d * (t - txt[i] * h) + txt[i + M]) % q;

                if (t < 0)
                    t = (t + q);
            }
        }
}

int main()
{
        char txt[] = "Porn Videos";
        char pat[] = "Porn";
        int q = 101;
        search(pat, txt, q);
        return 0;
}
```