Job Scheduling:-

```
#include <bits/stdc++.h>

using namespace std;

void display(int arr[]){
    for(int i=0;i<5;i++){
        cout<<arr[i]<<" ";
    }
}
void display(string arr[], int arr1[]){
    int profit = 0;
    for(int i=0;i<4;i++){
        cout<<arr[i]<<" ";
        profit+=arr1[i];
    }

    cout<<endl<<"Profit: "<<profit;

}



void sort(int arr1[], string arr2[], int arr3[]){
    for(int i=0;i<5;i++){
        for(int j=i+1;j<5;j++){
            if(arr1[i]<arr1[j]){
                int temp1=arr1[i];
                arr1[i]=arr1[j];
                arr1[j]=temp1;

                string temp2=arr2[i];
                arr2[i]=arr2[j];
                arr2[j]=temp2;

                int temp3=arr3[i];
                arr3[i]=arr3[j];
                arr3[j]=temp3;
            }
        }
    }
}

int main()
{
    int arr_profit[5] = {100,20,30,50,80};
    string arr_job[5] = {"J1","J2","J3","J4","J5"};
    int arr_deadline[5] = {3,2,4,1,2};
    int job = 4;
    string result[job] = {"0","0","0","0"};

    sort(arr_profit,arr_job, arr_deadline);

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

        int n = arr_deadline[i];
        for(int j=n-1;j>=0;j--){
            if(result[j]=="0"){
                result[j]=arr_job[i];
                break;
            }
        }

    }
    cout<<"Result: ";
    display(result, arr_profit);




    return 0;
}
```

N queen:-

```
#include<bits/stdc++.h>
using namespace std;

bool isSafe(int **arr, int x, int y, int n){
    for(int row=0;row<x;row++){
        if(arr[row][y]==1){
            return false;
        }
    }

    int row =x;
    int col =y;
    while(row>=0 && col>=0){
        if(arr[row][col]==1){
            return false;
        }
        row--;
        col--;
    }

    row =x;
    col =y;
    while(row>=0 && col<n){
        if(arr[row][col]==1){
            return false;
        }
        row--;
        col++;
    }

    return true;
}

bool nQueen(int** arr, int x, int n){

    if(x>=n){
        return true;
    }

    for(int col=0;col<n;col++){
        if(isSafe(arr,x,col,n)){
            arr[x][col]=1;
            if(nQueen(arr,x+1,n)){
                return true;
            }
            arr[x][col]=0;
        }
    }
    return false;
}


int main(){
    int n;
    cin >> n;

    int *arr = new int[n];
    for(int i=0;i<n;i++){
        arr[i] = new int[n];
        for(int j=0;j<n;j++){
            arr[i][j]=0;
        }
    }

    if(nQueen(arr,0,n)){
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                cout << arr[i][j] << " ";
            }
            cout << endl;
        }
    }
    return 0;
}
```

kruskals:-

```
#include <bits/stdc++.h>
using namespace std;

// DSU data structure
// path compression + rank by union
class DSU {
    int* parent;
    int* rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    // Find function
    int find(int i)
    {
        if (parent[i] == -1)
```

```
            return i;

        return parent[i] = find(parent[i]);
    }

    // Union function
    void unite(int x, int y)
    {
        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
            }
            else if (rank[s1] > rank[s2]) {
                parent[s2] = s1;
            }
            else {
                parent[s2] = s1;
                rank[s1] += 1;
            }
        }
    }
};

class Graph {
    vector<vector<int> > edgelist;
    int V;

public:
    Graph(int V) { this->V = V; }

    // Function to add edge in a graph
    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });
    }

    void kruskals_mst()
    {
        // Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        cout << "Following are the edges in the "
                "constructed MST"
             << endl;
        for (auto edge : edgelist) {
            int w = edge[0];
            int x = edge[1];
            int y = edge[2];

            // Take this edge in MST if it does
            // not forms a cycle
            if (s.find(x) != s.find(y)) {
                s.unite(x, y);
                ans += w;
                cout << x << " -- " << y << " == " << w
                     << endl;
            }
        }
        cout << "Minimum Cost Spanning Tree: " << ans;
    }
};

// Driver code
int main()
{
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);

    // Function call
    g.kruskals_mst();

    return 0;
}
```

prims:-

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t"
             << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];
    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
    // vertex.
    key[0] = 0;

    // First node is always root of MST
    parent[0] = -1;

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent
            // vertices of m mstSet[v] is false for vertices
            // not yet included in MST Update the key only
            // if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false
                && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

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

// Driver's code
int main()
{
    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 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

Bfsdfs:-

```
#include<iostream>
#include<list>
#include<map>
#include<queue>

using namespace std;

class Graph{
    public:

    map<int, list<int>> adjList;
    map<int, bool> visited;
    queue<int> q;

    //Copy Constructor
    // Graph(const Graph &g){
    // }

    void addEdge(int src, int dest){
        adjList[src].push_back(dest);
        adjList[dest].push_back(src);
    }

    void DFS(int node){
        //Mark Node as visited
        visited[node] = true;
        //Print Node
        cout << node << " ";
        //Vist its neighbours and recurse
        for(int i : adjList[node]){
            //If node is not visited
            if(!visited[i]) DFS(i);
        }
    }

    void BFS(){
        //Recurse until queue is empty
        if(q.empty()) return;
        //Pop and assign 1st node in queue
        int node = q.front();
        q.pop();
        //Print node
        cout << node << " ";
        for(int i : adjList[node]){
            //If neighbour is not visited add it to queue
            if(!visited[i]){
                visited[i] = true;
                q.push(i);
            }
        }
        BFS();
    }
};



int main(){
    Graph g;
    g.addEdge(0,1);
    g.addEdge(0,2);
    g.addEdge(0,3);
    g.addEdge(1,3);
    g.addEdge(3,4);
    g.addEdge(4,5);
    g.addEdge(2,6);

    int ch;
    cout << "Enter 0 to perform DFS and 1 to perform BFS: ";
    cin >> ch;
    if(!ch){
        cout << "DFS on the given graph is :";
        g.DFS(0);
    }
    else{
        cout << "BFS on the given graph is: ";
        g.q.push(0);
        g.visited[0] = true;
        g.BFS();
    }
    return 0;
}
```

A*:-

```
#include<iostream>
#include<cmath>
#include<limits.h>

using namespace std;

//A* alogrithm to solve 8 puzzle problem

//Global variable to keep track of number of moves taken
int g = 0;

void Print(int puzzle[]){
    for(int i = 0; i < 9; i++){
        if(i % 3 == 0) cout << '\n';
        if(puzzle[i] == -1) cout << "_ ";
        else cout << puzzle[i] << " ";
    }
    cout << "\n\n";
}

void moveLeft(int start[], int position){
    swap(start[position], start[position - 1]);
}

void moveRight(int start[], int position){
    swap(start[position], start[position + 1]);
}

void moveUp(int start[], int position){
    swap(start[position], start[position - 3]);
}
```

```
void moveDown(int start[], int position){
    swap(start[position], start[position + 3]);
}


void Copy(int temp[], int real[]){
    for(int i = 0; i < 9; i++) temp[i] = real[i];
}


/*
For every number find difference in position in goal state and inital state
Difference in vertical + difference in horizontal i.e Manhattan Distance
*/
int heuristic(int start[], int goal[]){
    int h = 0;
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            if (start[i] == goal[j] && start[i] != -1){
                h += abs((j - i) / 3) + abs((j - i) % 3);
            }
        }
    }
    return h + g;
}


void moveTile(int start[], int goal[]){
    int emptyAt = 0;
    for(int i = 0; i < 9; i++){
        if(start[i] == -1){
            emptyAt = i;
            break;
        }
    }

    int t1[9],t2[9],t3[9],t4[9],f1 = INT_MAX,f2 = INT_MAX,f3 = INT_MAX,f4 =
INT_MAX;
    Copy(t1, start);
    Copy(t2, start);
    Copy(t3, start);
    Copy(t4, start);

    int row = emptyAt / 3;
    int col = emptyAt % 3;
    if(col - 1 >= 0){
        moveLeft(t1, emptyAt);
        f1 = heuristic(t1, goal);
    }

    if(col + 1 < 3){
        moveRight(t2, emptyAt);
        f2 = heuristic(t2, goal);
    }

    if(row + 1 < 3){
        moveDown(t3, emptyAt);
        f3 = heuristic(t3, goal);
    }

    if(row - 1 >= 0){
        moveUp(t4, emptyAt);
        f4 = heuristic(t4, goal);
    }


    //Find Least Heuristic State and Make the Move
    if(f1 <= f2 && f1 <= f3 && f1 <= f4 ){
        moveLeft(start, emptyAt);
    }
    else if(f2 <= f1 && f2 <= f3 && f2 <= f4 ){
        moveRight(start, emptyAt);
    }
    else if(f3 <= f1 && f3 <= f2 && f3 <= f4 ){
        moveDown(start, emptyAt);
    }
    else{
        moveUp(start, emptyAt);
    }
}

void solveEight(int start[], int goal[]){
    g++;
    //Move Tile
    moveTile(start, goal);
    Print(start);
    //Get Heuristic Value
    int f = heuristic(start, goal);
    if(f == g){
        cout << "Solved in " << f << " moves\n";
        return;
    }
    solveEight(start, goal);
}

/*
Count the number of inversion
If odd then unsolvable
else solvable
*/

bool solvable(int start[]){
    int invrs = 0;
    for(int i = 0; i < 9; i++){
        //1 2 3 -1 4 6 7 5 8

        if(start[i] <= 1) continue;
        for(int j = i + 1; j < 9; j++){
            if(start[j] == -1) continue;
            if(start[i] > start[j]) invrs++;
        }
    }
    return invrs & 1 ? false : true;
}

// void printImpossible(){
//     cout << R"(|￣￣￣￣￣￣￣￣￣￣￣|
//  ITS IMPOSSIBLE TO SOLVE
// |＿＿＿＿＿＿＿＿＿＿＿|
//    (\__/)  ||
//    (•ㅅ•)  ||
//    /       づ
// )" << '\n';
// }

int main(){
    int start[9];
    int goal[9];
    cout << "Enter the start state:(Enter -1 for empty):";
    for(int i = 0; i < 9; i++){
        cin >> start[i];
    }
    cout << "Enter the goal state:(Enter -1 for empty):";
    for(int i = 0; i < 9; i++){
        cin >> goal[i];
    }

    // verify if possible to solve
    Print(start);
    if(solvable(start)) solveEight(start, goal);
    else cout << "\nImpossible To Solve\n";

    return 0;
}

/*
Test Cases
1 2 3 -1 4 6 7 5 8
1 2 3 4 5 6 7 8 -1

1 2 3 4 8 -1 7 6 5
1 2 3 4 5 6 7 8 -1


1 2 3 4 5 6 -1 8 7
1 2 3 4 5 6 7 8 -1

*/
```

selection sort:-

```
def selectionSort(arr):
    for i in range(len(arr)):
        min = float('-inf')
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                arr[i],arr[j] = arr[j], arr[i]
    return arr

print(selectionSort([89,56,45,34,65,76]))
```