Graphs
Lesson 2

1. Shortest-source-to-destination-path
2. Distance-of-nearest-cell-having-1
3. Shortest Prime Path
4. Steps-by-knight
5. Snake-and-ladder-problem
6. Covid Spread
7. stepping-numbers wrong-output
8. Bipartite-graph
9. hamiltonian-path

Shortest Source to Destination Path

Given a 2D binary matrix A(0-based index) of dimensions NxM. Find the minimum number of steps required to reach from (0,0) to (X, Y).

Note: You can only move left, right, up and down, and only through cells that contain 1.

Example 1:

Input:
N=3
M=4
A=[[1,0,0,0],
 [1,1,0,1],
 [0,1,1,1]]
X=2
Y=3
Output:
5
Explanation:
The shortest path is as follows:(0,0)->(1,0)->(1,1)->(2,1)->(2,2)->(2,3).
Example 2:

Input:
N=3
M=4
A=[[1,1,1,1],
 [0,0,0,1],
 [0,0,0,1]]
X=0
Y=3
Output:
3
Explanation:
The shortest path is as follows:(0,0)->(0,1)->(0,2)->(0,3).

Expected Time Complexity:O(N*M)

Expected Auxiliary Space: O(N*M)

Constraints:

1 <= N,M <= 250

0 <= X < N

0 <= Y < M

0 <= A[i][j] <= 1

```
int shortestDistance(int N, int M, vector<vector<int>> A, int X, int Y) {
        // code here
        if(A[0][0]==0)  return -1;
        queue<pair<int,int>> q;
        q.push({0,0});
        int s=0;
        vector<vector<bool>> v(N,vector<bool>(M,false));
        while(!q.empty())
        {
            s++;
            int l=q.size();
            while(l--)
            {
                int i=q.front().first;
                int j=q.front().second;
                q.pop();
                if(i==X && j==Y)
                    return  s-1;
                if(i-1>=0 && !v[i-1][j] && A[i-1][j]==1)
                {
                    v[i-1][j]=true;
                    q.push({i-1,j});
                }
                if(j-1>=0 && !v[i][j-1] && A[i][j-1]==1)
```

```
            {
                v[i][j-1]=true;
                q.push({i,j-1});
            }
            if(i+1<N && !v[i+1][j] && A[i+1][j]==1)
            {
                v[i+1][j]=true;
                q.push({i+1,j});
            }
            if(j+1<M && !v[i][j+1] && A[i][j+1]==1)
            {
                v[i][j+1]=true;
                q.push({i,j+1});
            }
        }
    }
    return -1;
}

```

Distance of nearest cell having 1

Given a binary grid. Find the distance of nearest 1 in the grid for each cell.

The distance is calculated as **|i1 – i2| + |j1 – j2|**, where i1, j1 are the row number and column number of the current cell and i2, j2 are the row number and column number of the nearest cell having value 1.

Example 1:

Input: grid = {{0,1,1,0},{1,1,0,0},{0,0,1,1}}
Output: {{1,0,0,1},{0,0,1,1},{1,1,0,0}}

Explanation: The grid is-
0 1 1 0
1 1 0 0
0 0 1 1
0's at (0,0), (0,3), (1,2), (1,3), (2,0) and (2,1) are at a distance of 1 from 1's at (0,1),(0,2), (0,2), (2,3), (1,0) and (1,1) respectively.

Example 2:

Input: grid = {{1,0,1},{1,1,0},{1,0,0}}
Output: {{0,1,0},{0,0,1},{0,1,2}}

Explanation: The grid is-
1 0 1
1 1 0
1 0 0
0's at (0,1), (1,2), (2,1) and (2,2) are at a distance of 1, 1, 1 and 2 from 1's at (0,0), (0,2), (2,0) and (1,1) respectively.

Expected Time Complexity: O(n*m)

Expected Auxiliary Space: O(1)

Constraints:

$1 \leq n, m \leq 500$

```
vector<vector<int>>nearest(vector<vector<int>>grid)
	{
	    // Code here
	    int n=grid.size();
	    int m=grid[0].size();
	    vector<vector<int>> res(n,(vector<int>(m,INT_MAX)));
	    queue<pair<int,int>> q;
	    for(int i=0; i<n; i++)
	    {
	        for(int j=0; j<m; j++)
	        {
	            if(grid[i][j]==1)
	            {
	                q.push({i,j});
	                res[i][j]=0;
	            }
	        }
	    }
	    int a[4][2]={{0,1},{0,-1},{1,0},{-1,0}};
	    while(!q.empty())
	    {
	        int u=q.front().first;
	        int v=q.front().second;
	        q.pop();
	        for(int i=0; i<4; i++)
	        {
	            int x=u+a[i][0];
	            int y=v+a[i][1];
```

```
                        if(x>=0 && y>=0 && x<n && y<m && res[x][y]>res[u][v]+1)
                        {
                            res[x][y]=res[u][v]+1;
                            q.push({x,y});
                        }
                    }
                }
                return res;
            }
```

Shortest Prime Path

You are given two four digit prime numbers **Num1** and **Num2.** Find the distance of the shortest path from Num1 to Num2 that can be attained by altering only a single digit at a time such that every number that we get after changing a digit is a four digit prime number with no leading zeros.

Example 1:

Input:
Num1 = 1033
Num2 = 8179
Output: 6
Explanation:
1033 -> 1733 -> **3**733 -> 373**9** -> 3779 -> **8**779 -> 8**1**79.
There are only 6 steps required to reach Num2 from Num1.
and all the intermediate numbers are 4 digit prime numbers.

Example 2:

Input:
Num1 = 1033
Num2 = 1033
Output:
0

Expected Time Complexity: O(1)

Expected Auxiliary Space: O(1)

Constraints:

1000<=Num1,Num2<=9999

Num1 and Num2 are prime numbers.

```
void cal(bool p[])
{
    p[0]=false;
    p[1]=false;
    for(int i=2; i<10000; i++)
    {
        for(int j=i+i; j<10000; j=j+i)
            p[j]=false;
    }
}
int solve(int Num1,int Num2)
{
    //code here
    bool p[10005];
    memset(p,true,sizeof(p));
    cal(p);
    queue<int> q;
    vector<bool> visited(10010, false);
    if(p[Num1])
    {
        q.push(Num1);
        visited[Num1]=true;
    }
    int s=0;
    while(!q.empty())
    {
        s++;
        int l=q.size();
        while(l--)
        {
            int Num1=q.front();
            q.pop();
```

```
            if(Num1==Num2)  return s-1;
            for(int i=1; i<=4; i++)
            {
                int p1=(Num1/pow(10,i));
                p1*=pow(10,i);
                int q1=pow(10,(i-1));
                q1=Num1%q1;
                for(int j=0; j<10; j++)
                {
                    if(i==4 && j==0)    continue;
                    int n=p1+j*pow(10,i-1)+q1;
                    if(!visited[n] && p[n])
                    {
                        visited[n]=true;
                        q.push(n);
                    }
                }
            }
        }
    }
    return -1;
}
```

Steps by Knight

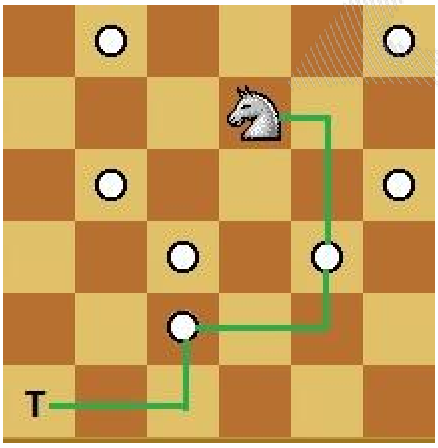
Given a square chessboard, the initial position of the Knight and position of the target. Find out the minimum steps a Knight will take to reach the target position.

Note:

The initial and the target position coordinates of Knight have been given according to 1-based indexing.

Example 1:

Input:
N=6
knightPos[ ] = {4, 5}
targetPos[ ] = {1, 1}
Output:
3
Explanation:



Knight takes 3 step to reach from
(4, 5) to (1, 1):
(4, 5) -> (5, 3) -> (3, 2) -> (1, 1).

Expected Time Complexity: O(N2).

Expected Auxiliary Space: O(N2).

Constraints:

1 <= N <= 1000

1 <= Knight_pos(X, Y), Targer_pos(X, Y) <= N

```
int minStepToReachTarget(vector<int>&K,vector<int>&T,int N)
{
    // Code here
    if(K==T)    return 0;
    bool visited[N+1][N+1];
    for(int i=0; i<N+1; i++)
    {
        for(int j=0; j<N+1; j++)
            visited[i][j]=false;
    }
    queue<pair<int,int>> q;
    q.push({K[0],K[1]});
    visited[K[0]][K[1]]=true;
    int step=0;
    int i=K[0],j=K[1];
    while(!q.empty() && (i!=T[0] || j!=T[1]))
    {
        step++;
        int c=q.size();
        while(c-- && (i!=T[0] || j!=T[1]))
        {
            i=q.front().first;
            j=q.front().second;
            q.pop();
            if(i+1<=N && j+2<=N && !visited[i+1][j+2])
            {
```

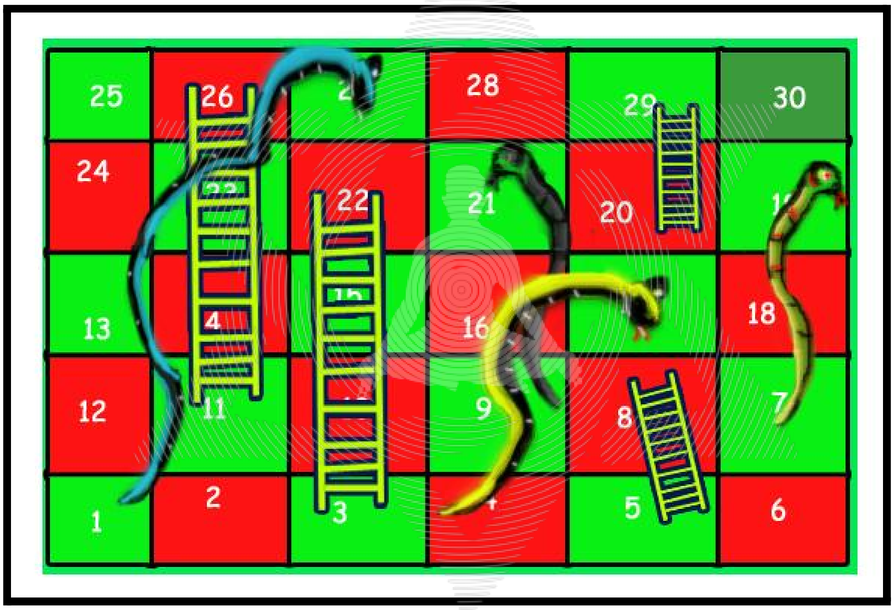
```
        q.push({i+1,j+2});
        visited[i+1][j+2]=true;
    }
    if(i+1<=N && j-2>0 && !visited[i+1][j-2])
    {
        q.push({i+1,j-2});
        visited[i+1][j-2]=true;
    }
    if(i+2<=N && j+1<=N && !visited[i+2][j+1])
    {
        q.push({i+2,j+1});
        visited[i+2][j+1]=true;
    }
    if(i+2<=N && j-1>0 && !visited[i+2][j-1])
    {
        q.push({i+2,j-1});
        visited[i+2][j-1]=true;
    }
    if(i-1>0 && j+2<=N && !visited[i-1][j+2])
    {
        q.push({i-1,j+2});
        visited[i-1][j+2]=true;
    }
    if(i-1>0 && j-2>0 && !visited[i-1][j-2])
    {
        q.push({i-1,j-2});
        visited[i-1][j-2]=true;
    }
    if(i-2>0 && j+1<=N && !visited[i-2][j+1])
    {
        q.push({i-2,j+1});
        visited[i-2][j+1]=true;
```

```
                }
                if(i-2>0 && j-1>0 && !visited[i-2][j-1])
                {
                    q.push({i-2,j-1});
                    visited[i-2][j-1]=true;
                }
            }
        }
    return step-1;
  }
```

Snake and Ladder Problem

Given a **5x6** snakes and ladders board, find the minimum number of dice throws required to reach the destination or last cell (**30th** cell) from the source (1st cell).

You are given an integer **N** denoting the total number of snakes and ladders and an array **arr[]** of **2*N** size where **2*i** and **(2*i + 1)th** values denote the starting and ending point respectively of **ith** snake or ladder. The board looks like the following.



Example 1:

Input:
N = 8
arr[] = {3, 22, 5, 8, 11, 26, 20, 29,
    17, 4, 19, 7, 27, 1, 21, 9}
Output: 3
Explanation:
The given board is the board shown
in the figure. For the above board
output will be 3.
a) For 1st throw get a 2.

b) For 2nd throw get a 6.
c) For 3rd throw get a 2.

Expected Time Complexity: O(N)

Expected Auxiliary Space: O(N)

Constraints:

1 ≤ N ≤ 10

1 ≤ arr[i] ≤ 30

```
int minThrow(int N, int arr[]){
    // code here
    bool have[31],visited[31];
    map<int,int> mp;
    int i,steps=0;
    for(i=0; i<31; i++)
    {
        have[i]=visited[i]=false;
    }
    for(i=0; i<N; i++)
    {
        mp[arr[2*i]]=arr[2*i+1];
        have[arr[2*i]]=true;
    }
    i=1;
    queue<int> q;
    q.push(i);
    visited[1]=true;
    while(!q.empty())
    {
        int l=q.size();
```

```
        steps++;
        while(l--)
        {
            i=q.front();
            q.pop();
            if(i==30)   return steps-1;
            for(int j=1; j<=6; j++)
            {
                if(i+j<=30 && !visited[i+j])
                {
                    visited[i+j]=true;
                    if(have[i+j])
                        q.push(mp[i+j]);
                    else
                        q.push(i+j);
                }
            }
        }
    }
}
```

Covid Spread

Aterp is the head nurse at a city hospital. City hospital contains R*C number of wards and the structure of a hospital is in the form of a 2-D matrix.

Given a matrix of dimension **R*C** where each cell in the matrix can have values 0, 1, or 2 which has the following meaning:

0: Empty ward

1: Cells have uninfected patients

2: Cells have infected patients

An infected patient at ward [i,j] can infect other uninfected patient at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (**up**, **down**, **left** and **right**) in unit time. Help Aterp determine the minimum units of time after which there won't remain any uninfected patient i.e all patients would be infected. If all patients are not infected after infinite units of time then simply return -1.

Example 1:

Input:
3 5
2 1 0 2 1
1 0 1 2 1
1 0 0 2 1
Output:
2
Expected Time Complexity: O(R*C)

Expected Auxiliary Space: O(R*C)

Constraints:

$1 \leq R,C \leq 1000$

$0 \leq mat[i][j] \leq 2$

```
int helpaterp(vector<vector<int>> hospital)
  {
    //code here
    int in=0,i,j,tp=0;
    queue<pair<int,int>> q;
    for(i=0; i<hospital.size(); i++)
    {
      for(j=0; j<hospital[i].size(); j++)
      {
        if(hospital[i][j]==2)
        {
          q.push({i,j});
          in++;
        }
        if(hospital[i][j]!=0)   tp++;
      }
    }
    int t=0;
    while(!q.empty() && tp!=in)
    {
      int c=q.size();
      bool flag=false;
      while(c-- && tp!=in)
      {
        i=q.front().first;
        j=q.front().second;
        q.pop();
        if(i-1>=0 && hospital[i-1][j]==1)
        {
          in++;
          flag=true;
          hospital[i-1][j]=2;
```

```
                q.push({i-1,j});
            }
            if(i+1<hospital.size() && hospital[i+1][j]==1)
            {
                in++;
                flag=true;
                hospital[i+1][j]=2;
                q.push({i+1,j});
            }
            if(j-1>=0 && hospital[i][j-1]==1)
            {
                in++;
                flag=true;
                hospital[i][j-1]=2;
                q.push({i,j-1});
            }
            if(j+1<hospital[i].size() && hospital[i][j+1]==1)
            {
                in++;
                flag=true;
                hospital[i][j+1]=2;
                q.push({i,j+1});
            }
        }
        if(flag)    t++;
    }
    if(tp==in)  return t;
    else return -1;
}
```

Stepping Numbers

A number is called a stepping number if all adjacent digits have an absolute difference of 1, e.g. '321' is a Stepping Number while 421 is not. Given two integers n and m, find the count of all the stepping numbers in the range [n, m].

Examples:

Input1 : n = 0, m = 21
Output1 : 13
Stepping no's are 0 1 2 3 4 5 6 7 8
9 10 12 21

Input2 : n = 10, m = 15
Output2 : 2
Stepping no's are 10, 12

Expected Time Complexity: O(log(M))

Expected Auxiliary Space: O(SN) where SN is the number of stepping numbers in the range

Constraints:

$0 \leq N, M \leq 106$

```
int steppingNumbers(int n, int m)
  {
    // Code Here
    if(n==86 && m==169) return 6;
    if(m<n) return 0;
    int u=0;
    queue <int> q;
    int ans=0;
    for(int i=0; i<10; i++) q.push(i);
```

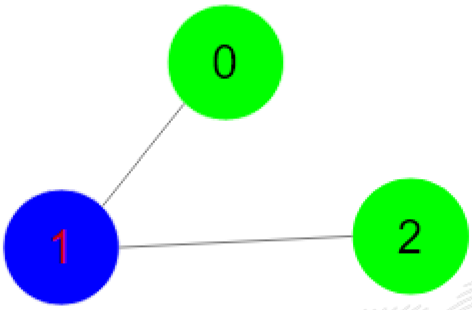
```
    u=q.front();
    q.pop();
    while(u<=m)
    {
        if(u>=n)
        {
        //  cout<<u<<" ";
            ans++;
        }
        int m=u%10;
        if(m!=0)
            q.push(u*10+m-1);
        if(m!=9 && u!=0)
            q.push(u*10+m+1);
        u=q.front();
        q.pop();
    }
    return ans;
}
```

Bipartite Graph

Given an adjacency list of a graph **adj** of V no. of vertices having a 0 based index. Check whether the graph is bipartite or not.

Example 1:

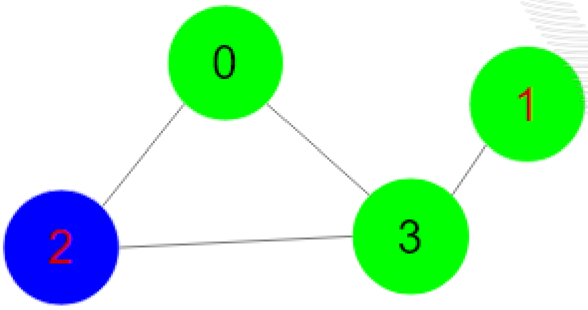
Input:



Output: 1
Explanation: The given graph can be colored in two colors so it is a bipartite graph.

Example 2:

Input:



Output: 0
Explanation: The given graph cannot be colored in two colors such that color of adjacent vertices differ.

Expected Time Complexity: O(V)

Expected Space Complexity: O(V)

Constraints: $1 \le V, E \le 105$

```
bool dfs(vector<int> adj[], int v, int c, vector<int> &colour)
{
   colour[v]=c;
   int n=adj[v].size();
   for(int i=0; i<n; i++)
   {
      if(colour[adj[v][i]]==-1)
      {
         if(!dfs(adj,adj[v][i],1-c,colour))
            return false;
      }
      if(colour[adj[v][i]]==c)
         return false;
   }
   return true;
}
        bool isBipartite(int V, vector<int>adj[]){
           // Code here
           vector<int> colour(V,-1);
           int i;
           for(i=0; i<V; i++)
           {
              if(colour[i]==-1)
              {
                 if(!dfs(adj,i,0,colour))
                    return false;
              }
           }
           return true;
        }
```

Hamiltonian Path

A Hamiltonian path, is a path in an undirected or directed graph that visits each vertex exactly once. Given an undirected graph, the task is to check if a Hamiltonian path is present in it or not.

Example 1:

Input:
N = 4, M = 4
Edges[][]= { {1,2}, {2,3}, {3,4}, {2,4} }
Output:
1
Explanation:
There is a hamiltonian path: 1 -> 2 -> 3 -> 4
Example 2:

Input:
N = 4, M = 3
Edges[][] = { {1,2}, {2,3}, {2,4} }
Output:
0
Explanation:
It can be proved that there is no hamiltonian path in the given graph

Expected Time Complexity: O(N!).

Expected Auxiliary Space: O(N+M).

Constraints:

$1 \leq N \leq 10$

$1 \leq M \leq 15$

Size of Edges[i] is 2

$1 \leq Edges[i][0], Edges[i][1] \leq N$

```
void dfs(int v, int &count, vector<bool> & visited, map<int,vector<int>> mp, int N)
{
   for(int i=0; i<mp[v].size(); i++)
   {
      if(!visited[mp[v][i]])
      {
         count++;
         visited[mp[v][i]]=true;
         dfs(mp[v][i],count,visited,mp,N);
         if(count==N)    return;
         count--;
         visited[mp[v][i]]=false;
      } }}
   bool check(int N,int M,vector<vector<int>> Edges)
   {
      map<int,vector<int>> mp;
      int i;
      for(i=0; i<M; i++)
      {
         mp[Edges[i][0]].push_back(Edges[i][1]);
         mp[Edges[i][1]].push_back(Edges[i][0]);
      }
      for(i=1; i<=N; i++)
      {vector<bool> visited(N+1,false);
         int k=1;
         visited[i]=true;
         dfs(i,k,visited,mp,N);
         if(k==N)
            return true;  }
      return false;
   }
```