AI LAB-EXERCISES

PAKHI VASHISHTH

201B171

**Problem 1: Solve “Tower of Hanoi” with only 2 disks and then 3 disks.**

Program: #include <bits/stdc++.h>

using namespace std;

void towerOfHanoi(int n, char from\_rod,

                    char to\_rod, char aux\_rod)

{

    if (n == 0)

    {

        return;

    }

    towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

    cout << "Move disk " << n << " from rod " << from\_rod <<

                                " to rod " << to\_rod << endl;

    towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

int main()

{

    int n = 3;

    towerOfHanoi(n, 'A', 'B', 'C');

    return 0;

}

**Problem 2: Solve “4-Queens” and “8-Queens” puzzle.**

#include <bits/stdc++.h>

#define N 4

using namespace std;

void printSolution(int board[N][N])

{

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++)

            cout << " " << 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) {

        cout << "Solution does not exist";

        return false;

    }

    printSolution(board);

    return true;

}

int main()

{

    solveNQ();

    return 0;

}

**Problem 3: Solve “4-color map” problem.**

#include<stdio.h>

int G[50][50],x[50]; *//G:adjacency matrix,x:colors*

void next\_color(int k){

   int i,j;

   x[k]=1; *//coloring vertex with color1*

   for(i=0;i<k;i++){ *//checking all k-1 vertices-backtracking*

     if(G[i][k]!=0 && x[k]==x[i]) *//if connected and has same color*

       x[k]=x[i]+1; *//assign higher color than x[i]*

   } }

int main(){

  int n,e,i,j,k,l;

  printf("Enter no. of vertices : ");

  scanf("%d",&n);

  printf("Enter no. of edges : ");

  scanf("%d",&e);

   for(i=0;i<n;i++)

    for(j=0;j<n;j++)

      G[i][j]=0; *//assign 0 to all index of adjacency matrix*

  printf("Enter indexes where value is 1-->\n");

  for(i=0;i<e;i++){

    scanf("%d %d",&k,&l);

    G[k][l]=1;

    G[l][k]=1;

  }

  for(i=0;i<n;i++)

 next\_color(i); *//coloring each vertex*

 printf("Colors of vertices -->\n");

  for(i=0;i<n;i++) *//displaying color of each vertex*

    printf("Vertex[%d] : %d\n",i+1,x[i]);

  return 0; }

**Problem 4: Solve “8 – puzzle” and “15-puzzle” take any initial and goal state.**

#include <bits/stdc++.h>

using namespace std;

#define N 3

*// state space tree nodes*

struct *Node*

{

*// stores the parent node of the current node*

*// helps in tracing path when the answer is found*

*Node*\* parent;

*// stores matrix*

    int mat[N][N];

*// stores blank tile coordinates*

    int x, y;

*// stores the number of misplaced tiles*

    int cost;

*// stores the number of moves so far*

    int level;

};

*// Function to print N x N matrix*

int printMatrix(int mat[N][N])

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

            printf("%d ", mat[i][j]);

        printf("\n");

    }

}

*// Function to allocate a new node*

*Node\** newNode(int mat[N][N], int x, int y, int newX,

              int newY, int level, *Node\** parent)

{

*Node*\* node = new *Node*;

*// set pointer for path to root*

    node->parent = parent;

*// copy data from parent node to current node*

    memcpy(node->mat, mat, sizeof node->mat);

*// move tile by 1 position*

    swap(node->mat[x][y], node->mat[newX][newY]);

*// set number of misplaced tiles*

    node->cost = INT\_MAX;

*// set number of moves so far*

    node->level = level;

*// update new blank tile coordinates*

    node->x = newX;

    node->y = newY;

    return node;

}

*// bottom, left, top, right*

int row[] = { 1, 0, -1, 0 };

int col[] = { 0, -1, 0, 1 };

*// Function to calculate the number of misplaced tiles*

*// ie. number of non-blank tiles not in their goal position*

int calculateCost(int initial[N][N], int final[N][N])

{

    int count = 0;

    for (int i = 0; i < N; i++)

      for (int j = 0; j < N; j++)

        if (initial[i][j] && initial[i][j] != final[i][j])

           count++;

    return count;

}

*// Function to check if (x, y) is a valid matrix coordinate*

int isSafe(int x, int y)

{

    return (x >= 0 && x < N && y >= 0 && y < N);

}

*// print path from root node to destination node*

void printPath(*Node\** root)

{

    if (root == NULL)

        return;

    printPath(root->parent);

    printMatrix(root->mat);

    printf("\n");

}

*// Comparison object to be used to order the heap*

struct *comp*

{

    bool operator()(*const* *Node\** lhs, *const* *Node\** rhs) *const*

    {

        return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);

    }

};

*// Function to solve N\*N - 1 puzzle algorithm using*

*// Branch and Bound. x and y are blank tile coordinates*

*// in initial state*

void solve(int initial[N][N], int x, int y,

           int final[N][N])

{

*// Create a priority queue to store live nodes of*

*// search tree;*

    priority\_queue<*Node*\*, std::vector<*Node*\*>, *comp*> pq;

*// create a root node and calculate its cost*

*Node*\* root = newNode(initial, x, y, x, y, 0, NULL);

    root->cost = calculateCost(initial, final);

*// Add root to list of live nodes;*

    pq.push(root);

*// Finds a live node with least cost,*

*// add its childrens to list of live nodes and*

*// finally deletes it from the list.*

    while (!pq.empty())

    {

*// Find a live node with least estimated cost*

*Node*\* min = pq.top();

*// The found node is deleted from the list of*

*// live nodes*

        pq.pop();

*// if min is an answer node*

        if (min->cost == 0)

        {

*// print the path from root to destination;*

            printPath(min);

            return;

        }

*// do for each child of min*

*// max 4 children for a node*

        for (int i = 0; i < 4; i++)

        {

            if (isSafe(min->x + row[i], min->y + col[i]))

            {

*// create a child node and calculate*

*// its cost*

*Node*\* child = newNode(min->mat, min->x,

                              min->y, min->x + row[i],

                              min->y + col[i],

                              min->level + 1, min);

                child->cost = calculateCost(child->mat, final);

*// Add child to list of live nodes*

                pq.push(child);

            }

        }

    }

}

*// Driver code*

int main()

{

*// Initial configuration*

*// Value 0 is used for empty space*

    int initial[N][N] =

    {

        {1, 2, 3},

        {5, 6, 0},

        {7, 8, 4}

    };

*// Solvable Final configuration*

*// Value 0 is used for empty space*

    int final[N][N] =

    {

        {1, 2, 3},

        {5, 8, 6},

        {0, 7, 4}

    };

*// Blank tile coordinates in initial*

*// configuration*

    int x = 1, y = 2;

    solve(initial, x, y, final);

    return 0;

}

**Problem 5: Solve “Latin Square” problem.**

#include<stdio.h>

void printLatin(int n)

{

    int k = n+1;

    for (int i=1; i<=n; i++)

    {

        int temp = k;

        while (temp <= n)

        {

            printf("%d ", temp);

            temp++;

        }

        for (int j=1; j<k; j++)

            printf("%d ", j);

        k--;

        printf("\n");

    }

}

int main(void)

{

    int n = 5;

    printLatin(n);

    return 0;

}

**Problem 6: Code the game: Tick-Tack-Toe.**

#include<bits/stdc++.h>

using namespace std;

#define COMPUTER 1

#define HUMAN 2

#define SIDE 3 *// Length of the board*

*// Computer will move with 'O'*

*// and human with 'X'*

#define COMPUTERMOVE 'O'

#define HUMANMOVE 'X'

*// A function to show the current board status*

void showBoard(char board[][SIDE])

{

    printf("\n\n");

    printf("\t\t\t %c | %c | %c \n", board[0][0],

                            board[0][1], board[0][2]);

    printf("\t\t\t--------------\n");

    printf("\t\t\t %c | %c | %c \n", board[1][0],

                            board[1][1], board[1][2]);

    printf("\t\t\t--------------\n");

    printf("\t\t\t %c | %c | %c \n\n", board[2][0],

                            board[2][1], board[2][2]);

    return;

}

*// A function to show the instructions*

void showInstructions()

{

    printf("\t\t\t Tic-Tac-Toe\n\n");

    printf("Choose a cell numbered from 1 to 9 as below"

            " and play\n\n");

    printf("\t\t\t 1 | 2 | 3 \n");

    printf("\t\t\t--------------\n");

    printf("\t\t\t 4 | 5 | 6 \n");

    printf("\t\t\t--------------\n");

    printf("\t\t\t 7 | 8 | 9 \n\n");

    printf("-\t-\t-\t-\t-\t-\t-\t-\t-\t-\n\n");

    return;

}

*// A function to initialise the game*

void initialise(char board[][SIDE], int moves[])

{

*// Initiate the random number generator so that*

*// the same configuration doesn't arises*

    srand(time(NULL));

*// Initially the board is empty*

    for (int i=0; i<SIDE; i++)

    {

        for (int j=0; j<SIDE; j++)

            board[i][j] = ' ';

    }

*// Fill the moves with numbers*

    for (int i=0; i<SIDE\*SIDE; i++)

        moves[i] = i;

*// randomise the moves*

    random\_shuffle(moves, moves + SIDE\*SIDE);

    return;

}

*// A function to declare the winner of the game*

void declareWinner(int whoseTurn)

{

    if (whoseTurn == COMPUTER)

        printf("COMPUTER has won\n");

    else

        printf("HUMAN has won\n");

    return;

}

*// A function that returns true if any of the row*

*// is crossed with the same player's move*

bool rowCrossed(char board[][SIDE])

{

    for (int i=0; i<SIDE; i++)

    {

        if (board[i][0] == board[i][1] &&

            board[i][1] == board[i][2] &&

            board[i][0] != ' ')

            return (true);

    }

    return(false);

}

*// A function that returns true if any of the column*

*// is crossed with the same player's move*

bool columnCrossed(char board[][SIDE])

{

    for (int i=0; i<SIDE; i++)

    {

        if (board[0][i] == board[1][i] &&

            board[1][i] == board[2][i] &&

            board[0][i] != ' ')

            return (true);

    }

    return(false);

}

*// A function that returns true if any of the diagonal*

*// is crossed with the same player's move*

bool diagonalCrossed(char board[][SIDE])

{

    if (board[0][0] == board[1][1] &&

        board[1][1] == board[2][2] &&

        board[0][0] != ' ')

        return(true);

    if (board[0][2] == board[1][1] &&

        board[1][1] == board[2][0] &&

        board[0][2] != ' ')

        return(true);

    return(false);

}

*// A function that returns true if the game is over*

*// else it returns a false*

bool gameOver(char board[][SIDE])

{

    return(rowCrossed(board) || columnCrossed(board)

            || diagonalCrossed(board) );

}

*// A function to play Tic-Tac-Toe*

void playTicTacToe(int whoseTurn)

{

*// A 3\*3 Tic-Tac-Toe board for playing*

    char board[SIDE][SIDE];

    int moves[SIDE\*SIDE];

*// Initialise the game*

    initialise(board, moves);

*// Show the instructions before playing*

    showInstructions();

    int moveIndex = 0, x, y;

*// Keep playing till the game is over or it is a draw*

    while (gameOver(board) == false &&

            moveIndex != SIDE\*SIDE)

    {

        if (whoseTurn == COMPUTER)

        {

            x = moves[moveIndex] / SIDE;

            y = moves[moveIndex] % SIDE;

            board[x][y] = COMPUTERMOVE;

            printf("COMPUTER has put a %c in cell %d\n",

                    COMPUTERMOVE, moves[moveIndex]+1);

            showBoard(board);

            moveIndex ++;

            whoseTurn = HUMAN;

        }

        else if (whoseTurn == HUMAN)

        {

            x = moves[moveIndex] / SIDE;

            y = moves[moveIndex] % SIDE;

            board[x][y] = HUMANMOVE;

            printf ("HUMAN has put a %c in cell %d\n",

                    HUMANMOVE, moves[moveIndex]+1);

            showBoard(board);

            moveIndex ++;

            whoseTurn = COMPUTER;

        }

    }

*// If the game has drawn*

    if (gameOver(board) == false &&

            moveIndex == SIDE \* SIDE)

        printf("It's a draw\n");

    else

    {

*// Toggling the user to declare the actual*

*// winner*

        if (whoseTurn == COMPUTER)

            whoseTurn = HUMAN;

        else if (whoseTurn == HUMAN)

            whoseTurn = COMPUTER;

*// Declare the winner*

        declareWinner(whoseTurn);

    }

    return;

}

*// Driver program*

int main()

{

*// Let us play the game with COMPUTER starting first*

    playTicTacToe(COMPUTER);

    return (0);

}

**Problem 7: Code the game: Checkers.**

#include<stdio.h>

#define ROWS 8

#define COLS 8

#define EMPTY 1

#define RED 2

#define BLACK 3

#define REDKING 4

#define BLACKKING 5

#define ISRED(c)  (c == RED || c == REDKING)

#define ISBLACK(c) (c == BLACK || c == BLACKKING))

#define ISEMPTY(c) (c == 1)

void printDisplay(int d[][COLS]);

void swapIJKL(int d[ROWS][COLS], int i, int j, int k, int l);

char value2symbol(int i);

void printDisplayFancy(int d[][COLS]);

int Playersturn(int d[][COLS], int player,int i,int j,int k,int l);

void printDisplayFancy(int d[][COLS])

{

    int rr, cc, pp;

    printf("  +---+---+---+---+---+---+---+---+\n");

    for (rr=0; rr<ROWS; ++rr)

    {

        printf("%d |", rr+1);

        for (cc=0; cc<COLS; ++cc)

        {

            printf(" %c |", value2symbol(d[rr][cc]) );

        }

        printf("\n");

        printf("  +---+---+---+---+---+---+---+---+\n");

    }

    printf("    a   b   c   d   e   f   g   h\n");

}

void swapIJKL(int d[ROWS][COLS], int i, int j, int k, int l)

{

    int temp;

    printf("SWAP: %d,%d to %d,%d\n", i, j, k, l);

    temp = d[i][j];

    d[i][j] = d[k][l];

    d[k][l] = temp;

}

char value2symbol(int i)

{

    switch(i)

    {

    case 0:

            return ' ';

        case 1:

            return 'E';

        case 2:

            return '$';

        case 3:

            return '@';

    }

    return ('?');

}

int Playersturn(int d[][COLS], int player,int i,int j,int k,int l)

{

    int jmp\_r;

    int jmp\_c;

    if(player == RED){

        printf("RED move from %d,%d to %d,%d\n", i, j, k, l);

    } else {

        printf("BLACK move from %d,%d to %d,%d\n", i, j, k, l);

    }

    if(i < 0 && ROWS <= i){

        printf("i is out of bounds\n");

        return -1;

    }

    if(j < 0 && COLS <= j){

        printf("j is out of bound");

        return -1;

    }

    if(k < 0 && ROWS <= k){

        printf("k is out of bounds");

        return -1;

    }

    if(l < 0 && COLS<= l){

        printf("l is out of bounds\n");

        return -1;

    }

    if(player == RED){

        if(d[i][j] != RED){

            printf("move your own piece!\n");

            return -1;

        }

    } else {

        if(d[i][j] != BLACK){

            printf("move your own piece!\n");

            return -1;

        }

    }

    if(d[k][l] != EMPTY){

        printf("You must move to a empty location");

        return -1;

    }

    if(player == RED){

        if(i >= k){

            printf("RED player must move down\n");

            return -1;

        }

    } else {

        if(i <= k){

            printf("BLACK player must move up\n");

            return -1;

        }

    }

    if(i - k == -1 || i - k == 1){

        if(j - l == 1 || j - l == -1){

            swapIJKL(d,i,j,k,l);

            return 0;

        }

    }

    if(i - k == -2 || i - k == 2){

        if(j - l == -2 || j - l == 2){

            if(i < k){

                jmp\_r = i + 1;

            } else {

                jmp\_r = i - 1;

            }

            if(j < l){

                jmp\_c = j + 1;

            } else {

                jmp\_c = j - 1;

            }

            if(player==RED && d[jmp\_r][jmp\_c]!=BLACK)

            {

                printf("Enemeny is not Black at %d%d",jmp\_r, jmp\_c);

                return -1;

            }

            if(player==BLACK && d[jmp\_r][jmp\_c] != RED){

                printf("you can only jump over an enemy player\n");

                return -1;

            }

            d[jmp\_r][jmp\_c] = 1;

            swapIJKL(d,i,j,k,l);

            return 0;

        }

    }

    printf("You can only move diagnally\n");

    return -1;

}

int main()

{

    int r,c;

    int pr, pb;

    int i, k;

    char j, l;

    int d[ROWS][COLS]={

    {0,2,0,2,0,2,0,2},

    {2,0,2,0,2,0,2,0},

    {0,2,0,2,0,2,0,2},

    {1,0,1,0,1,0,1,0},

    {0,1,0,1,0,1,0,1},

    {3,0,3,0,3,0,3,0},

    {0,3,0,3,0,3,0,3},

    {3,0,3,0,3,0,3,0}};

    for(;;){

        printDisplayFancy(d);

        while(1){

            printf("Red's turn: ");

            scanf("%d%c",&i,&j);

            printf("to: ");

            scanf("%d%c",&k,&l);

            if(Playersturn(d, RED, i-1,j - 'a',k-1,l - 'a') == 0)

                break;

            printf("Illegal move, try again\n");

        }

        printDisplayFancy(d);

        while(1){

            printf("Black's turn: ");

            scanf("%d%c",&i,&j);

            printf("to :");

            scanf("%d%c",&k,&l);

            if(Playersturn(d, BLACK, i-1,j - 'a',k-1,l - 'a') == 0)

                break;

            printf("Illegal move, try again\n");

        }

    }

    getchar();

    getchar();

    return 0;

}

**Problem 8: Write a program to solve water jug problem.**

“You are given an m liter jug and a n liter jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d is less than n.”

#include<bits/stdc++.h>

using namespace std;

int x;

int y;

void show(int a, int b);

int min(int w, int z)

{

if (w < z)

return w;

else

return z;

}

void show(int a, int b)

{

cout << setw(12) << a << setw(12) << b<<endl;

}

void s(int n)

{

int xq = 0, yq = 0;

int t;

cout << setw(15) <<"FIRST JUG"<< setw(15) <<"SECOND JUG"<<endl;

while (xq != n && yq!=n )

{

  if (xq == 0)

  {

    xq = x;

    show(xq, yq);

  }

  else if (yq == y)

  {

    yq = 0;

    show(xq, yq);

  }

  else

  {

    t = min(y - yq, xq);

    yq= yq + t;

    xq = xq - t;

    show(xq, yq);

      }

}

}

int main()

{

int n;

cout << "Enter the liters of water required out of the two jugs: ";

cin >> n;

cout << "Enter the capacity of the first jug: ";

cin >> x;

cout << "Enter the capacity of the second jug: ";

cin >> y;

if(n<x || n<y)

{ if(n%(\_\_gcd(x,y))==0)

    s(n);

  else

  cout<<"This is not possible....\n";

}

else

  cout<<"This is not possible....\n";

}

**Problem 9: Write a Program to Implement Missionaries-Cannibals Problems.**

“In this problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, that the missionaries present on the bank cannot be outnumbered by cannibals. The boat cannot cross the river by itself with no people on board.”

#include<iostream>

#include<iomanip>

using namespace std;

class *game*{

public:

int counto, i;

char left[6], right[6];

int m\_num, c\_num;

bool side;

int ml\_count, cl\_count;

int mr\_count, cr\_count;

game(){

counto = 1;

ml\_count = cl\_count = 3;

mr\_count = cr\_count = 0;

side = false;

for (i = 0; i<3; i++){

left[i] = 'M';

left[i + 3] = 'C';

right[i] = ' ';

right[i + 3] = ' ';

}

}

void get(){

start:

cout << "\nEnter no.of missionaries= ";

cin >> m\_num;

cout << "\nEnter no.of cannibals= ";

cin >> c\_num;

if (m\_num>3 || c\_num>3 || m\_num<0 || c\_num<0)

goto start;

else if ((m\_num + c\_num)>2 || (m\_num + c\_num == 0))

goto start;

}

void displaymc(){

cout << "\nleft side\tright side\n";

for (i = 0; i<3; i++)

cout << left[i] << " ";

cout << "\t\t";

for (i = 0; i<3; i++)

cout << right[i] << " ";

cout << endl;

for (i = 3; i<6; i++)

cout << left[i] << " ";

cout << "\t\t";

for (i = 3; i<6; i++)

cout << right[i] << " ";

cout << endl;

if (counto % 2 == 0){

side = true;

cout << "\nBoat on right side of river\n";

}

else{

side = false;

cout << "\nBoat on left side of river\n";

}

}

void boat\_lr(){

for (i = 0; i<m\_num; i++){

if (left[0] == 'M'){

left[0] = ' ';

right[0] = 'M';

ml\_count -= 1;

mr\_count += 1;

}

else if (left[1] == 'M'){

left[1] = ' ';

right[1] = 'M';

ml\_count -= 1;

mr\_count += 1;

}

else if (left[2] == 'M'){

left[2] = ' ';

right[2] = 'M';

ml\_count -= 1;

mr\_count += 1;

}

}

for (i = 0; i<c\_num; i++){

if (left[3] == 'C'){

left[3] = ' ';

right[3] = 'C';

cl\_count -= 1;

cr\_count += 1;

}

else if (left[4] == 'C'){

left[4] = ' ';

right[4] = 'C';

cl\_count -= 1;

cr\_count += 1;

}

else if (left[5] == 'C'){

left[5] = ' ';

right[5] = 'C';

cl\_count -= 1;

cr\_count += 1;

}

}

}

void boat\_rl(){

for (i = 0; i<m\_num; i++){

if (right[0] == 'M'){

right[0] = ' ';

left[0] = 'M';

ml\_count += 1;

mr\_count -= 1;

}

else if (right[1] == 'M'){

right[1] = ' ';

left[1] = 'M';

ml\_count += 1;

mr\_count -= 1;

}

else if (right[2] == 'M'){

right[2] = ' ';

left[2] = 'M';

ml\_count += 1;

mr\_count -= 1;

}

}

for (i = 0; i<c\_num; i++){

if (right[3] == 'C'){

right[3] = ' ';

left[3] = 'C';

cl\_count += 1;

cr\_count -= 1;

}

else if (right[4] == 'C'){

right[4] = ' ';

left[4] = 'C';

cl\_count += 1;

cr\_count -= 1;

}

else if (right[5] == 'C'){

right[5] = ' ';

left[5] = 'C';

cl\_count += 1;

cr\_count -= 1;

}

}

}

};

int main(){

*game* g;

while (true){

if (g.ml\_count<g.cl\_count && g.ml\_count>0 || g.mr\_count<g.cr\_count && g.mr\_count>0){

cout << "\n\n~~~~~~~~~~~~~~~~YOU LOST!!~~~~~~~~~~~~~~~~\n";

break;

}

else if (g.cr\_count == g.mr\_count && g.cr\_count == 3 && g.mr\_count == 3){

cout << "\n\n~~~~~~~~~~~~~~~~YOU WON!!~~~~~~~~~~~~~~~~\n";

break;

}

else{

g.displaymc();

g.get();

if (g.side == false)

g.boat\_lr();

else

g.boat\_rl();

}

g.counto++;

}

return 0;

}

**Problem 10: Write a program to solve traveling salesman problem.**

“Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.”

#include <bits/stdc++.h>

using namespace std;

#define V 4

int travllingSalesmanProblem(int graph[][V], int s)

{

    vector<int> vertex;

    for (int i = 0; i < V; i++)

        if (i != s)

            vertex.push\_back(i);

    int min\_path = INT\_MAX;

    do {

        int current\_pathweight = 0;

        int k = s;

        for (int i = 0; i < vertex.size(); i++) {

            current\_pathweight += graph[k][vertex[i]];

            k = vertex[i];

        }

        current\_pathweight += graph[k][s];

        min\_path = min(min\_path, current\_pathweight);

    } while (

        next\_permutation(vertex.begin(), vertex.end()));

    return min\_path;

}

int main()

{

    int graph[][V] = { { 0, 10, 15, 20 },

                    { 10, 0, 35, 25 },

                    { 15, 35, 0, 30 },

                    { 20, 25, 30, 0 } };

    int s = 0;

    cout << travllingSalesmanProblem(graph, s) << endl;

    return 0;

}

**Problem 11: Write a Program to Implement Camel and Banana Puzzle.**

“A person has 3000 bananas and a camel. The person wants to transport the maximum number of bananas to a destination which is 1000 KMs away, using only the camel as a mode of transportation. The camel cannot carry more than 1000 bananas at a time and eats a banana every km it travels. What is the maximum number of bananas that can be transferred to the destination using only camel (no other mode of transportation is allowed).”

#include <bits/stdc++.h>

using namespace std;

int dp[1001][3001];

int recBananaCnt(int A, int B, int C)

{

    if (B <= A) {

        return 0;

    }

    if (B <= C) {

        return B - A;

    }

    if (A == 0) {

        return B;

    }

    if (dp[A][B] != -1) {

        return dp[A][B];

    }

    int maxCount = INT\_MIN;

    int tripCount = B % C == 0 ? ((2 \* B) / C) - 1

                            : ((2 \* B) / C) + 1;

    for (int i = 1; i <= A; i++) {

        int curCount

            = recBananaCnt(A - i,

                        B - tripCount \* i, C);

        if (curCount > maxCount) {

            maxCount = curCount;

            dp[A][B] = maxCount;

        }

    }

    return maxCount;

}

int maxBananaCnt(int A, int B, int C)

{

    memset(dp, -1, sizeof(dp));

    return recBananaCnt(A, B, C);

}

int main()

{

    int A = 1000;

    int B = 3000;

    int C = 1000;

    cout << maxBananaCnt(A, B, C);

    return 0;

}

**Problem 12: Write a program of Magic Square.**

“A magic square of order n is an arrangement of n2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant.”

#include <stdio.h>

#include <string.h>

void generateSquare(int n)

{

    int magicSquare[n][n];

    memset(magicSquare, 0, sizeof(magicSquare));

    int i = n / 2;

    int j = n - 1;

    for (int num = 1; num <= n \* n;) {

        if (i == -1 && j == n)

        {

            j = n - 2;

            i = 0;

        }

        else {

            if (j == n)

                j = 0;

            if (i < 0)

                i = n - 1;

        }

        if (magicSquare[i][j])

        {

            j -= 2;

            i++;

            continue;

        }

        else

            magicSquare[i][j] = num++;

        j++;

        i--;

    }

    printf("The Magic Square for n=%d:\nSum of "

        "each row or column %d:\n\n",

        n, n \* (n \* n + 1) / 2);

    for (i = 0; i < n; i++) {

        for (j = 0; j < n; j++)

            printf("%3d ", magicSquare[i][j]);

        printf("\n");

    }

}

int main()

{

    int n = 7;

    generateSquare(n);

    return 0;

}

**Problem 13: Knight’s Tour problem using Backtracking.**

“Given an N\*N board with the Knight placed on the first block of an empty board. Moving according to the rules of chess knight must visit each square exactly once. Print the order of each cell in which they are visited.”

#include <stdio.h>

#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],

                int xMove[], int yMove[]);

int isSafe(int x, int y, int sol[N][N])

{

    return (x >= 0 && x < N && y >= 0 && y < N

            && sol[x][y] == -1);

}

void printSolution(int sol[N][N])

{

    for (int x = 0; x < N; x++) {

        for (int y = 0; y < N; y++)

            printf(" %2d ", sol[x][y]);

        printf("\n");

    }

}

int solveKT()

{

    int sol[N][N];

    for (int x = 0; x < N; x++)

        for (int y = 0; y < N; y++)

            sol[x][y] = -1;

    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };

    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    sol[0][0] = 0;

    if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {

        printf("Solution does not exist");

        return 0;

    }

    else

        printSolution(sol);

    return 1;

}

int solveKTUtil(int x, int y, int movei, int sol[N][N],

                int xMove[N], int yMove[N])

{

    int k, next\_x, next\_y;

    if (movei == N \* N)

        return 1;

    for (k = 0; k < 8; k++) {

        next\_x = x + xMove[k];

        next\_y = y + yMove[k];

        if (isSafe(next\_x, next\_y, sol)) {

            sol[next\_x][next\_y] = movei;

            if (solveKTUtil(next\_x, next\_y, movei + 1, sol,

                            xMove, yMove)

                == 1)

                return 1;

            else

                sol[next\_x][next\_y] = -1;

        }

    }

    return 0;

}

int main()

{

    solveKT();

    return 0;

}

**Problem 14: Code the game software: Chess.**

#include <iostream>

#include <stdlib.h>

class *GamePiece*

{

public:

    GamePiece(char PieceColor) : mPieceColor(PieceColor) {}

    ~GamePiece() {}

*virtual* char GetPiece() = 0;

    char GetColor() {

        return mPieceColor;

    }

    bool IsLegalMove(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

*GamePiece*\* qpDest = GameBoard[iDestRow][iDestCol];

        if ((qpDest == 0) || (mPieceColor != qpDest->GetColor())) {

            return AreSquaresLegal(iSrcRow, iSrcCol, iDestRow, iDestCol, GameBoard);

        }

        return false;

    }

private:

*virtual* bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) = 0;

    char mPieceColor;

};

class *PawnPiece* : public *GamePiece*

{

public:

    PawnPiece(char PieceColor) : *GamePiece*(PieceColor) {}

    ~PawnPiece() {}

private:

*virtual* char GetPiece() {

        return 'P';

    }

    bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

*GamePiece*\* qpDest = GameBoard[iDestRow][iDestCol];

        if (qpDest == 0) {

*// Destination square is unoccupied*

            if (iSrcCol == iDestCol) {

                if (GetColor() == 'W') {

                    if (iDestRow == iSrcRow + 1) {

                        return true;

                    }

                } else {

                    if (iDestRow == iSrcRow - 1) {

                        return true;

                    }

                }

            }

        } else {

*// Dest holds piece of opposite color*

            if ((iSrcCol == iDestCol + 1) || (iSrcCol == iDestCol - 1)) {

                if (GetColor() == 'W') {

                    if (iDestRow == iSrcRow + 1) {

                        return true;

                    }

                } else {

                    if (iDestRow == iSrcRow - 1) {

                        return true;

                    }

                }

            }

        }

        return false;

    }

};

class *KnightPiece* : public *GamePiece*

{

public:

    KnightPiece(char PieceColor) : *GamePiece*(PieceColor) {}

    ~KnightPiece() {}

private:

*virtual* char GetPiece() {

        return 'N';

    }

    bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

*// Destination square is unoccupied or occupied by opposite color*

        if ((iSrcCol == iDestCol + 1) || (iSrcCol == iDestCol - 1)) {

            if ((iSrcRow == iDestRow + 2) || (iSrcRow == iDestRow - 2)) {

                return true;

            }

        }

        if ((iSrcCol == iDestCol + 2) || (iSrcCol == iDestCol - 2)) {

            if ((iSrcRow == iDestRow + 1) || (iSrcRow == iDestRow - 1)) {

                return true;

            }

        }

        return false;

    }

};

class *BishopPiece* : public *GamePiece*

{

public:

    BishopPiece(char PieceColor) : *GamePiece*(PieceColor) {}

    ~BishopPiece() {}

private:

*virtual* char GetPiece() {

        return 'B';

    }

    bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

        if ((iDestCol - iSrcCol == iDestRow - iSrcRow) || (iDestCol - iSrcCol == iSrcRow - iDestRow)) {

*// Make sure that all invervening squares are empty*

            int iRowOffset = (iDestRow - iSrcRow > 0) ? 1 : -1;

            int iColOffset = (iDestCol - iSrcCol > 0) ? 1 : -1;

            int iCheckRow;

            int iCheckCol;

            for (iCheckRow = iSrcRow + iRowOffset, iCheckCol = iSrcCol + iColOffset;

                iCheckRow !=  iDestRow;

                iCheckRow = iCheckRow + iRowOffset, iCheckCol = iCheckCol + iColOffset)

            {

                if (GameBoard[iCheckRow][iCheckCol] != 0) {

                    return false;

                }

            }

            return true;

        }

        return false;

    }

};

class *RookPiece* : public *GamePiece*

{

public:

    RookPiece(char PieceColor) : *GamePiece*(PieceColor) {}

    ~RookPiece() {}

private:

*virtual* char GetPiece() {

        return 'R';

    }

    bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

        if (iSrcRow == iDestRow) {

*// Make sure that all invervening squares are empty*

            int iColOffset = (iDestCol - iSrcCol > 0) ? 1 : -1;

            for (int iCheckCol = iSrcCol + iColOffset; iCheckCol !=  iDestCol; iCheckCol = iCheckCol + iColOffset) {

                if (GameBoard[iSrcRow][iCheckCol] != 0) {

                    return false;

                }

            }

            return true;

        } else if (iDestCol == iSrcCol) {

*// Make sure that all invervening squares are empty*

            int iRowOffset = (iDestRow - iSrcRow > 0) ? 1 : -1;

            for (int iCheckRow = iSrcRow + iRowOffset; iCheckRow !=  iDestRow; iCheckRow = iCheckRow + iRowOffset) {

                if (GameBoard[iCheckRow][iSrcCol] != 0) {

                    return false;

                }

            }

            return true;

        }

        return false;

    }

};

class *QueenPiece* : public *GamePiece*

{

public:

    QueenPiece(char PieceColor) : *GamePiece*(PieceColor) {}

    ~QueenPiece() {}

private:

*virtual* char GetPiece() {

        return 'Q';

    }

    bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

        if (iSrcRow == iDestRow) {

*// Make sure that all invervening squares are empty*

            int iColOffset = (iDestCol - iSrcCol > 0) ? 1 : -1;

            for (int iCheckCol = iSrcCol + iColOffset; iCheckCol !=  iDestCol; iCheckCol = iCheckCol + iColOffset) {

                if (GameBoard[iSrcRow][iCheckCol] != 0) {

                    return false;

                }

            }

            return true;

        } else if (iDestCol == iSrcCol) {

*// Make sure that all invervening squares are empty*

            int iRowOffset = (iDestRow - iSrcRow > 0) ? 1 : -1;

            for (int iCheckRow = iSrcRow + iRowOffset; iCheckRow !=  iDestRow; iCheckRow = iCheckRow + iRowOffset) {

                if (GameBoard[iCheckRow][iSrcCol] != 0) {

                    return false;

                }

            }

            return true;

        } else if ((iDestCol - iSrcCol == iDestRow - iSrcRow) || (iDestCol - iSrcCol == iSrcRow - iDestRow)) {

*// Make sure that all invervening squares are empty*

            int iRowOffset = (iDestRow - iSrcRow > 0) ? 1 : -1;

            int iColOffset = (iDestCol - iSrcCol > 0) ? 1 : -1;

            int iCheckRow;

            int iCheckCol;

            for (iCheckRow = iSrcRow + iRowOffset, iCheckCol = iSrcCol + iColOffset;

                iCheckRow !=  iDestRow;

                iCheckRow = iCheckRow + iRowOffset, iCheckCol = iCheckCol + iColOffset)

            {

                if (GameBoard[iCheckRow][iCheckCol] != 0) {

                    return false;

                }

            }

            return true;

        }

        return false;

    }

};

class *KingPiece* : public *GamePiece*

{

public:

    KingPiece(char PieceColor) : *GamePiece*(PieceColor) {}

    ~KingPiece() {}

private:

*virtual* char GetPiece() {

        return 'K';

    }

    bool AreSquaresLegal(int iSrcRow, int iSrcCol, int iDestRow, int iDestCol, *GamePiece\** GameBoard[8][8]) {

        int iRowDelta = iDestRow - iSrcRow;

        int iColDelta = iDestCol - iSrcCol;

        if (((iRowDelta >= -1) && (iRowDelta <= 1)) &&

            ((iColDelta >= -1) && (iColDelta <= 1)))

        {

            return true;

        }

        return false;

    }

};

class *CBoard*

{

public:

    CBoard() {

        for (int iRow = 0; iRow < 8; ++iRow) {

            for (int iCol = 0; iCol < 8; ++iCol) {

                MainGameBoard[iRow][iCol] = 0;

            }

        }

*// Allocate and place black pieces*

        for (int iCol = 0; iCol < 8; ++iCol) {

            MainGameBoard[6][iCol] = new *PawnPiece*('B');

        }

        MainGameBoard[7][0] = new *RookPiece*('B');

        MainGameBoard[7][1] = new *KnightPiece*('B');

        MainGameBoard[7][2] = new *BishopPiece*('B');

        MainGameBoard[7][3] = new *KingPiece*('B');

        MainGameBoard[7][4] = new *QueenPiece*('B');

        MainGameBoard[7][5] = new *BishopPiece*('B');

        MainGameBoard[7][6] = new *KnightPiece*('B');

        MainGameBoard[7][7] = new *RookPiece*('B');

*// Allocate and place white pieces*

        for (int iCol = 0; iCol < 8; ++iCol) {

            MainGameBoard[1][iCol] = new *PawnPiece*('W');

        }

        MainGameBoard[0][0] = new *RookPiece*('W');

        MainGameBoard[0][1] = new *KnightPiece*('W');

        MainGameBoard[0][2] = new *BishopPiece*('W');

        MainGameBoard[0][3] = new *KingPiece*('W');

        MainGameBoard[0][4] = new *QueenPiece*('W');

        MainGameBoard[0][5] = new *BishopPiece*('W');

        MainGameBoard[0][6] = new *KnightPiece*('W');

        MainGameBoard[0][7] = new *RookPiece*('W');

    }

    ~CBoard() {

        for (int iRow = 0; iRow < 8; ++iRow) {

            for (int iCol = 0; iCol < 8; ++iCol) {

                delete MainGameBoard[iRow][iCol];

                MainGameBoard[iRow][iCol] = 0;

            }

        }

    }

    void Print() {

        using namespace std;

*const* int kiSquareWidth = 4;

*const* int kiSquareHeight = 3;

        for (int iRow = 0; iRow < 8\*kiSquareHeight; ++iRow) {

            int iSquareRow = iRow/kiSquareHeight;

*// Print side border with numbering*

            if (iRow % 3 == 1) {

                cout << '-' << (char)('1' + 7 - iSquareRow) << '-';

            } else {

                cout << "---";

            }

*// Print the chess board*

            for (int iCol = 0; iCol < 8\*kiSquareWidth; ++iCol) {

                int iSquareCol = iCol/kiSquareWidth;

                if (((iRow % 3) == 1) && ((iCol % 4) == 1 || (iCol % 4) == 2) && MainGameBoard[7-iSquareRow][iSquareCol] != 0) {

                    if ((iCol % 4) == 1) {

                        cout << MainGameBoard[7-iSquareRow][iSquareCol]->GetColor();

                    } else {

                        cout << MainGameBoard[7-iSquareRow][iSquareCol]->GetPiece();

                    }

                } else {

                    if ((iSquareRow + iSquareCol) % 2 == 1) {

                        cout << '\*';

                    } else {

                        cout << ' ';

                    }

                }

            }

            cout << endl;

        }

*// Print the bottom border with numbers*

        for (int iRow = 0; iRow < kiSquareHeight; ++iRow) {

            if (iRow % 3 == 1) {

                cout << "---";

                for (int iCol = 0; iCol < 8\*kiSquareWidth; ++iCol) {

                    int iSquareCol = iCol/kiSquareWidth;

                    if ((iCol % 4) == 1) {

                        cout << (iSquareCol + 1);

                    } else {

                        cout << '-';

                    }

                }

                cout << endl;

            } else {

                for (int iCol = 1; iCol < 9\*kiSquareWidth; ++iCol) {

                    cout << '-';

                }

                cout << endl;

            }

        }

    }

    bool IsInCheck(char PieceColor) {

*// Find the king*

        int iKingRow;

        int iKingCol;

        for (int iRow = 0; iRow < 8; ++iRow) {

            for (int iCol = 0; iCol < 8; ++iCol) {

                if (MainGameBoard[iRow][iCol] != 0) {

                    if (MainGameBoard[iRow][iCol]->GetColor() == PieceColor) {

                        if (MainGameBoard[iRow][iCol]->GetPiece() == 'K') {

                            iKingRow = iRow;

                            iKingCol = iCol;

                        }

                    }

                }

            }

        }

*// Run through the opponent's pieces and see if any can take the king*

        for (int iRow = 0; iRow < 8; ++iRow) {

            for (int iCol = 0; iCol < 8; ++iCol) {

                if (MainGameBoard[iRow][iCol] != 0) {

                    if (MainGameBoard[iRow][iCol]->GetColor() != PieceColor) {

                        if (MainGameBoard[iRow][iCol]->IsLegalMove(iRow, iCol, iKingRow, iKingCol, MainGameBoard)) {

                            return true;

                        }

                    }

                }

            }

        }

        return false;

    }

    bool CanMove(char PieceColor) {

*// Run through all pieces*

        for (int iRow = 0; iRow < 8; ++iRow) {

            for (int iCol = 0; iCol < 8; ++iCol) {

                if (MainGameBoard[iRow][iCol] != 0) {

*// If it is a piece of the current player, see if it has a legal move*

                    if (MainGameBoard[iRow][iCol]->GetColor() == PieceColor) {

                        for (int iMoveRow = 0; iMoveRow < 8; ++iMoveRow) {

                            for (int iMoveCol = 0; iMoveCol < 8; ++iMoveCol) {

                                if (MainGameBoard[iRow][iCol]->IsLegalMove(iRow, iCol, iMoveRow, iMoveCol, MainGameBoard)) {

*// Make move and check whether king is in check*

*GamePiece*\* qpTemp                   = MainGameBoard[iMoveRow][iMoveCol];

                                    MainGameBoard[iMoveRow][iMoveCol]   = MainGameBoard[iRow][iCol];

                                    MainGameBoard[iRow][iCol]           = 0;

                                    bool bCanMove = !IsInCheck(PieceColor);

*// Undo the move*

                                    MainGameBoard[iRow][iCol]           = MainGameBoard[iMoveRow][iMoveCol];

                                    MainGameBoard[iMoveRow][iMoveCol]   = qpTemp;

                                    if (bCanMove) {

                                        return true;

                                    }

                                }

                            }

                        }

                    }

                }

            }

        }

        return false;

    }

*GamePiece*\* MainGameBoard[8][8];

};

class *ChessBoard*

{

public:

    ChessBoard() : mcPlayerTurn('W') {}

    ~ChessBoard() {}

    void Start() {

        do {

            GetNextMove(mqGameBoard.MainGameBoard);

            AlternateTurn();

        } while (!IsGameOver());

        mqGameBoard.Print();

    }

    void GetNextMove(*GamePiece\** GameBoard[8][8]) {

        using namespace std;

        bool bValidMove     = false;

        do {

            system ("clear");

            cout<<endl<<endl<<"          Welcome to Chess Game Developed by Cppsecrets "<<endl<<endl<<endl;

            cout<<"                      Keys to sysmbols used "<<endl<<endl<<endl;

            cout<<" \* = white space"<<endl;

            cout<<" Blank space = black space"<<endl;

            cout<<" WP = White pawn &  BP = Black pawn"<<endl;

            cout<<" WN = White Knight & BN = Black Knight"<<endl;

            cout<<" WB = White Bishop & BB = Black Bishop"<<endl;

            cout<<" WR = White Rook & BR = Black Rook"<<endl;

            cout<<" WQ = White Queen & BQ = Black Queen"<<endl;

            cout<<" WK = White King & BK =Black King"<<endl;

            cout<<"Rule for move is :"<<endl;

            cout<<"Move by selecting row & column to another valid location using row & column"<<endl<<endl<<endl;

            mqGameBoard.Print();

*// Get input and convert to coordinates*

            cout << mcPlayerTurn << "'s Move: ";

            int iStartMove;

            cin >> iStartMove;

            int iStartRow = (iStartMove / 10) - 1;

            int iStartCol = (iStartMove % 10) - 1;

            cout << "To: ";

            int iEndMove;

            cin >> iEndMove;

            int iEndRow = (iEndMove / 10) - 1;

            int iEndCol = (iEndMove % 10) - 1;

*// Check that the indices are in range*

*// and that the source and destination are different*

            if ((iStartRow >= 0 && iStartRow <= 7) &&

                (iStartCol >= 0 && iStartCol <= 7) &&

                (iEndRow >= 0 && iEndRow <= 7) &&

                (iEndCol >= 0 && iEndCol <= 7)) {

*// Additional checks in here*

*GamePiece*\* qpCurrPiece = GameBoard[iStartRow][iStartCol];

*// Check that the piece is the correct color*

                if ((qpCurrPiece != 0) && (qpCurrPiece->GetColor() == mcPlayerTurn)) {

*// Check that the destination is a valid destination*

                    if (qpCurrPiece->IsLegalMove(iStartRow, iStartCol, iEndRow, iEndCol, GameBoard)) {

*// Make the move*

*GamePiece*\* qpTemp                   = GameBoard[iEndRow][iEndCol];

                        GameBoard[iEndRow][iEndCol]     = GameBoard[iStartRow][iStartCol];

                        GameBoard[iStartRow][iStartCol] = 0;

*// Make sure that the current player is not in check*

                        if (!mqGameBoard.IsInCheck(mcPlayerTurn)) {

                            delete qpTemp;

                            bValidMove = true;

                        } else { *// Undo the last move*

                            GameBoard[iStartRow][iStartCol] = GameBoard[iEndRow][iEndCol];

                            GameBoard[iEndRow][iEndCol]     = qpTemp;

                        }

                    }

                }

            }

            if (!bValidMove) {

                cout << "Invalid Move!" << endl;

            }

        } while (!bValidMove);

    }

    void AlternateTurn() {

        mcPlayerTurn = (mcPlayerTurn == 'W') ? 'B' : 'W';

    }

    bool IsGameOver() {

*// Check that the current player can move*

*// If not, we have a stalemate or checkmate*

        bool bCanMove(false);

        bCanMove = mqGameBoard.CanMove(mcPlayerTurn);

        if (!bCanMove) {

            if (mqGameBoard.IsInCheck(mcPlayerTurn)) {

                AlternateTurn();

                std::cout << "Checkmate, " << mcPlayerTurn << " Wins!" << std::endl;

            } else {

                std::cout << "Stalemate!" << std::endl;

            }

        }

        return !bCanMove;

    }

private:

*CBoard* mqGameBoard;

    char mcPlayerTurn;

};

int main() {

*ChessBoard* qGame;

    qGame.Start();

    return 0;

}

**Problem 15: K-Means Algorithm for Colour Compression.**

“One interesting application of clustering is in colour compression within images. For example,imagine you have an image with millions of colours. In most images, a large number of the colours will be unused, and many of the pixels in the image will have similar or even identical colours.”

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.image as img

from scipy import misc

def read\_image():

*# loading the png image as a 3d matrix*

    img = misc.imread('bird\_small.png')

*# uncomment the below code to view the loaded image*

*# plt.imshow(A) # plotting the image*

*# plt.show()*

*# scaling it so that the values are small*

    img = img / 255

    return img

def initialize\_means(img, clusters):

*# reshaping it or flattening it into a 2d matrix*

    points = np.reshape(img, (img.shape[0] \* img.shape[1],

                                            img.shape[2]))

    m, n = points.shape

*# clusters is the number of clusters*

*# or the number of colors that we choose.*

*# means is the array of assumed means or centroids.*

    means = np.zeros((clusters, n))

*# random initialization of means.*

    for i in range(clusters):

        rand1 = int(np.random.random(1)\*10)

        rand2 = int(np.random.random(1)\*8)

        means[i, 0] = points[rand1, 0]

        means[i, 1] = points[rand2, 1]

    return points, means

*# Function to measure the euclidean*

*# distance (distance formula)*

def distance(x1, y1, x2, y2):

    dist = np.square(x1 - x2) + np.square(y1 - y2)

    dist = np.sqrt(dist)

    return dist

def k\_means(points, means, clusters):

    iterations = 10 *# the number of iterations*

    m, n = points.shape

*# these are the index values that*

*# correspond to the cluster to*

*# which each pixel belongs to.*

    index = np.zeros(m)

*# k-means algorithm.*

    while(iterations > 0):

        for j in range(len(points)):

*# initialize minimum value to a large value*

            minv = 1000

            temp = None

            for k in range(clusters):

                x1 = points[j, 0]

                y1 = points[j, 1]

                x2 = means[k, 0]

                y2 = means[k, 1]

                if(distance(x1, y1, x2, y2) < minv):

                    minv = distance(x1, y1, x2, y2)

                    temp = k

                    index[j] = k

        for k in range(clusters):

            sumx = 0

            sumy = 0

            count = 0

            for j in range(len(points)):

                if(index[j] == k):

                    sumx += points[j, 0]

                    sumy += points[j, 1]

                    count += 1

            if(count == 0):

                count = 1

            means[k, 0] = float(sumx / count)

            means[k, 1] = float(sumy / count)

        iterations -= 1

    return means, index

def compress\_image(means, index, img):

*# recovering the compressed image by*

*# assigning each pixel to its corresponding centroid.*

    centroid = np.array(means)

    recovered = centroid[index.astype(int), :]

*# getting back the 3d matrix (row, col, rgb(3))*

    recovered = np.reshape(recovered, (img.shape[0], img.shape[1],

                                                    img.shape[2]))

*# plotting the compressed image.*

    plt.imshow(recovered)

    plt.show()

*# saving the compressed image.*

    misc.imsave('compressed\_' + str(clusters) +

                        '\_colors.png', recovered)

*# Driver Code*

if \_\_name\_\_ == '\_\_main\_\_':

    img = read\_image()

    clusters = 16

    clusters = int(input('Enter the number of colors in the compressed image. default = 16\n'))

    points, means = initialize\_means(img, clusters)

    means, index = k\_means(points, means, clusters)

    compress\_image(means, index, img)