Name – Indranil Bain
DEPT. – Computer Science & Technology

Enrolment No. – 2020CSB039, Assignment - 03
Subject – Artificial Intelligence LAB

Answer

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
#define ll long long
using namespace std;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int,int> pr;
typedef vector<pair<int,int> > vpr;
#define ff first
#define ss second
#define rep(n) for(int i = 0; i < n; i++)
#define repj(n) for(int j = 0; j < n; j++)
#define repk(n) for(int k = 0; k < n; k++)
#define fort(i,m,n) for(int i=m;i<n;i++)
#define fortj(j,m,n) for(int j=m;j<n;j++)
#define fora(arr,n) for(int i=0;i<n;i++) cin>>arr[i]
#define begud(k) cerr<<"\t-->"<<#k<<" = "<<k<<"\n";
#define endl "\n"
#define pb push_back
#define debug(args...) { string _debug_list = #args; replace(_debug_list.begin(), _debug_list.end(), ',', ' '); stringstream _debug_stream(_debug_list); istream_iterator<string> __it(_debug_stream); debug_func(__it, args); }
void debug_func(istream_iterator<string> _it) {
   cerr << "\n";
}
template <typename T, typename... Args>
void debug_func(istream_iterator<string> _it, T x, Args... args) {
   cerr << "[ " << *_it << " : " << x << " ]";
   cerr << " , ";
   debug_func(++_it, args...);
}
const int mod = 998244353;
ll power(ll a,ll b)
{
   if(b==0)
      return 1;
   ll res = power(a,b/2);
   ll ans = res*res;
    ans %= mod;
   if(b%2==0)
      return ans;
   return (a*ans)%mod;
}
bool sortbysec(const pair<string,int> &a,const pair<string,int> &b)
{
    return (a.second > b.second);
}

int fact(int n){
    if(n == 1 || n == 0)
        return 1;
    int ans = ((n % mod) * (fact(n - 1) % mod)) % mod;
    return ans;
}

bool compare(const pair<int, int> &p1, const pair<int, int> &p2){
    return p1.ff > p2.ff;
}

void solve(){
    int n, k = -1; // Size of the array
    cout << "Enter number of digits: ";
    cin >> n;
    vi a(n);
    cout << "Enter number: ";
    fora(a, n);
    for(int i = n - 2; i >=0; i--){
        if(a[i + 1] > a[i]){
            k = i;
            break;
        }

    }
    if(k == -1){
        cout << "Highest number only\n";
        return;
    }
    int l = INT_MAX, m = -1;
    fort(i, k + 1, n){
        if(a[i] > a[k]){
            l = min(l, a[i]);
            if(l == a[i]){
                m = i;
            }

        }
    }
    swap(a[k], a[m]);
    sort(a.begin() + k + 1, a.end());
    rep(n){
        cout << a[i];
    }
    cout << endl;

}
int32_t main()
{
    int t;
    cout << "Enter number of test cases: ";
    cin >> t;
    while(t--){
        solve();
    }
}
```

Answer

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
#define ll long long
using namespace std;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int,int> pr;
typedef vector<pair<int,int> > vpr;
#define ff first
#define ss second
#define rep(n) for(int i = 0; i < n; i++)
#define repj(n) for(int j = 0; j < n; j++)
#define repk(n) for(int k = 0; k < n; k++)
#define fort(i,m,n) for(int i=m;i<n;i++)
#define fortj(j,m,n) for(int j=m;j<n;j++)
#define fora(arr,n) for(int i=0;i<n;i++) cin>>arr[i]
#define begud(k) cerr<<"\t-->"<<#k<<" = "<<k<<"\n";
#define endl "\n"
#define pb push_back
#define debug(args...) { string _debug_list = #args; replace(_debug_list.begin(), _debug_list.end(), ',', ' '); stringstream _debug_stream(_debug_list); istream_iterator<string> __it(_debug_stream); debug_func(__it, args); }
void debug_func(istream_iterator<string> _it) {
   cerr << "\n";
}
template <typename T, typename... Args>
void debug_func(istream_iterator<string> _it, T x, Args... args) {
   cerr << "[ " << *_it << " : " << x << " ]";
   cerr << " , ";
   debug_func(++_it, args...);
}
const int mod = 998244353;
ll power(ll a,ll b)
{
   if(b==0)
      return 1;
   ll res = power(a,b/2);
   ll ans = res*res;
    ans %= mod;
   if(b%2==0)
      return ans;
   return (a*ans)%mod;
}
bool sortbysec(const pair<string,int> &a,const pair<string,int> &b)
{
    return (a.second > b.second);
}

int fact(int n){
    if(n == 1 || n == 0)
        return 1;
    int ans = ((n % mod) * (fact(n - 1) % mod)) % mod;
    return ans;
}

bool compare(const pair<int, int> &p1, const pair<int, int> &p2){
    return p1.ff > p2.ff;
}

void shift(int n, char src, char intermediate, char dest){
    if(!n){
        return;
    }
    shift(n - 1, src, dest, intermediate);
    cout << "Disc " << n << " moves from pole " << src << " to pole " << dest << endl;
    shift(n - 1, intermediate, src, dest);
}

void solve(){
    int n, k = -1; // Size of the array
    cout << "Enter number of discs: ";
    cin >> n;
    shift(n, 'A', 'B', 'C');
    cout << endl;

}
int32_t main()
{
    int t;
    cout << "Enter number of test cases: ";
    cin >> t;
    while(t--){
        solve();
    }
}
```

Answ

```cpp
#include <bits/stdc++.h>
using namespace std;
#define V 4

// implementation of traveling Salesman Problem
int travllingSalesmanProblem(int graph[][V], int s, vector<int>& path)
{
    // store all vertex apart from source vertex
    vector<int> vertex;
    for (int i = 0; i < V; i++)
        if (i != s)
            vertex.push_back(i);

    // store minimum weight Hamiltonian Cycle.
    int min_path = INT_MAX;
    do {

        // store current Path weight(cost)
        int current_pathweight = 0;

        // compute current path weight
        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];

        // update minimum
        if (current_pathweight < min_path) {
            min_path = current_pathweight;
            path = { s };
            for (int i : vertex)
                path.push_back(i);
            path.push_back(s);
        }

    } while (
        next_permutation(vertex.begin(), vertex.end()));

    return min_path;
}

// Driver Code
int main()
{
    // matrix representation of graph
    int graph[][V] = { { 0, 10, 15, 20 },
                       { 10, 0, 35, 25 },
                       { 15, 35, 0, 30 },
                       { 20, 25, 30, 0 } };
    int s = 0;
    vector<int> path;

    int minCost = travllingSalesmanProblem(graph, s, path);

    cout << "Optimal Path: ";
    for (int city : path) {
        cout << city + 1 << " ";
    }
    cout << "\nTotal Distance: " << minCost << endl;

    return 0;
}
```

Answer

```cpp
#include <cstdio>
#include <stack>
#include <map>
#include <algorithm>
using namespace std;

// Representation of a state (x, y)
// x and y are the amounts of water in litres in the two jugs respectively
struct state {
    int x, y;

    // Used by map to efficiently implement lookup of seen states
    bool operator < (const state& that) const {
        if (x != that.x) return x < that.x;
        return y < that.y;
    }
};

// Capacities of the two jugs respectively and the target amount
int capacity_x, capacity_y, target = 2;

void dfs(state start, stack <pair <state, int> >& path)   {
    stack <state> s;
    state goal = (state) {-1, -1};

    // Stores seen states so that they are not revisited and
    // maintains their parent states for finding a path through
    // the state space
    // Mapping from a state to its parent state and rule no. that
    // led to this state
    map <state, pair <state, int> > parentOf;

    s.push(start);
    parentOf[start] = make_pair(start, 0);

    while (!s.empty())    {
        // Get the state at the front of the stack
        state top = s.top();
        s.pop();

        // If the target state has been found, break
        if (top.y == target) {
            goal = top;
            break;
        }

        // Find the successors of this state
        // This step uses production rules to produce successors of the current state
        // while pruning away branches which have been seen before

        // Rule 1: (x, y) -> (capacity_x, y) if x < capacity_x
        // Fill the first jug
        if (top.x < capacity_x)  {
            state child = (state) {capacity_x, top.y};
            // Consider this state for visiting only if it has not been visited before
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 1);
            }
        }

        // Rule 2: (x, y) -> (x, capacity_y) if y < capacity_y
        // Fill the second jug
        if (top.y < capacity_y)  {
            state child = (state) {top.x, capacity_y};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 2);
            }
        }

        // Rule 3: (x, y) -> (0, y) if x > 0
        // Empty the first jug
        if (top.x > 0)  {
            state child = (state) {0, top.y};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 3);
            }
        }

        // Rule 4: (x, y) -> (x, 0) if y > 0
        // Empty the second jug
        if (top.y > 0)  {
            state child = (state) {top.x, 0};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 4);
            }
        }

        // Rule 5: (x, y) -> (min(x + y, capacity_x), max(0, x + y - capacity_x)) if y > 0
        // Pour water from the second jug into the first jug until the first jug is full
        // or the second jug is empty
        if (top.y > 0)  {
            state child = (state) {min(top.x + top.y, capacity_x), max(0, top.x + top.y - capacity_x)};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 5);
            }
        }

        // Rule 6: (x, y) -> (max(0, x + y - capacity_y), min(x + y, capacity_y)) if x > 0
        // Pour water from the first jug into the second jug until the second jug is full
        // or the first jug is empty
        if (top.x > 0)  {
            state child = (state) {max(0, top.x + top.y - capacity_y), min(top.x + top.y, capacity_y)};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 6);
            }
        }
    }

    // Target state was not found
    if (goal.x == -1 || goal.y == -1)
        return;

    // backtrack to generate the path through the state space
    path.push(make_pair(goal, 0));
    // remember parentOf[start] = (start, 0)
    while (parentOf[path.top().first].second != 0)
        path.push(parentOf[path.top().first]);
}

int main()  {
    stack <pair <state, int> > path;

    printf("Enter the capacities of the two jugs : ");
    scanf("%d %d", &capacity_x, &capacity_y);

    dfs((state) {0, 0}, path);
    if (path.empty())
        printf("\nTarget cannot be reached.\n");
    else    {
        printf("\nNumber of moves to reach the target : %ld\nOne path to the target is as follows :\n", path.size() - 1);
        while (!path.empty())   {
            state top = path.top().first;
            int rule = path.top().second;
            path.pop();

            switch (rule)   {
                case 0: printf("State : (%d, %d)\n#\n", top.x, top.y);
                        break;
                case 1: printf("State : (%d, %d)\nAction : Fill the first jug\n", top.x, top.y);
                        break;
                case 2: printf("State : (%d, %d)\nAction : Fill the second jug\n", top.x, top.y);
                        break;
                case 3: printf("State : (%d, %d)\nAction : Empty the first jug\n", top.x, top.y);
                        break;
                case 4: printf("State : (%d, %d)\nAction : Empty the second jug\n", top.x, top.y);
                        break;
                case 5: printf("State : (%d, %d)\nAction : Pour from second jug into first jug\n", top.x, top.y);
                        break;
                case 6: printf("State : (%d, %d)\nAction : Pour from first jug into second jug\n", top.x, top.y);
                        break;
            }
        }
    }

    return 0;
}
```

Answer

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <ctime>

using namespace std;

#define compareBoxes(box1, box2, box3) ((board[box1] == board[box2]) && (board[box2] == board[box3]) && (board[box1] != 0)) //Checkes if three items are the same, and makes sure they're not 0's.
#define numberToLetter(x) ((x > 0) ? (x == 1) ? 'X' : 'O' : ' ') //Takes the number and turns it into the letter or space.

int getWinner(int board[9]) {
   //Finds winner of game, if there is no winner, returns 0.
   int winner = 0;
   for (int x = 0; x < 3; x++) {
      if (compareBoxes(3*x, 3*x+1, 3*x+2)) { //Checks rows.
         winner = board[3*x];
         break;
      } else if (compareBoxes(x, x+3, x+6)) { //Checks columns.
         winner = board[x];
         break;

      } else if (compareBoxes(2*x, 4, 8-2*x) && (x < 2)) { //Checks diagonals. Doesn't check if x == 2.
         winner = board[4];
         break;
      }
   }
   return winner;
}
bool gameOver(int board[9]){
   //Checks if game is over, and announces who won, or if it was a tie.
   int winner = getWinner(board);
   if (winner > 0) {
      cout << numberToLetter(winner) << "wins!"<< endl;
      return true;
   }
   for (int x = 0; x < 9; x++) {
      if (board[x] == 0) return false;
   }
   cout << "Tie!\n\n";
   return true;
}

int willWin(int board[9], int player) {
   //Checks if a given player could win in the next plank.
   for (int x = 0; x < 9; x++) {
      int tempBoard[9];
      memcpy(tempBoard, board, 36);
      if (board[x] > 0) continue;
      tempBoard[x] = player;
      if(getWinner(tempBoard) == player) return x;
   }
   return -1;
}

int exceptionalCase(int board[9]) {
   //Finds bords that are exceptions to how the algorithm works.
   int cases[2][9] = {{1,0,0,0,2,0,0,0,1}, {0,1,0,1,2,0,0,0,0}}; //Boards that don't work with algorithe.
   int answers[2][4] = {{3,3,3,3}, {2,8,6,0}};
   int rotatedBoard[9] = {6,3,0,7,4,1,8,5,2};
   int newBoard[9];
   int tempBoard[9];
   for(int x = 0; x < 9; x++) {
      newBoard[x] = board[x];
   }
   for (int caseIndex = 0; caseIndex < 2; caseIndex++) {
      for(int rotation = 0; rotation < 4; rotation++) {
         for (int x = 0; x < 9; x++)
            tempBoard[x] = newBoard[x];

         int match = 0;
         //Rotates board so it works with different versions of the same board.
         for (int box = 0; box < 9; box++) {
            newBoard[box] = tempBoard[rotatedBoard[box]];
         }

         for (int x = 0; x < 9; x++) {
            if (newBoard[x] == cases[caseIndex][x]) match++;
            else break;
         }
         if (match == 9) return answers[caseIndex][rotation];
      }
   }
   return -1;
}

int getSpace(int board[9], int spaces[4]) {
   //Gets a random corner or side that's not taken.
   bool isSpaceEmpty = false;
   int y;
   for (int x = 0; x < 4; x++) {
      if (board[spaces[x]] == 0) {
         isSpaceEmpty = true;
         break;
      }
   }
   if (isSpaceEmpty) {
      do {
         y = rand() % 4;
      } while (board[spaces[y]] != 0);
      return spaces[y];
   }
   return -1;
}

void outputBoard(int board[9]) {
   for(int line = 0; line < 3; line++){
      for (int box = 0; box < 3; box++) {
         cout << numberToLetter(board[3*line+box]) << ((box < 2) ? '|' : '\n');
      }
      cout << ((line < 2) ? "-----\n" : "\n");
   }
}

int main(){
   int board[9] = {0,0,0,0,0,0,0,0,0}; //Starts empty board.
   int possibleWinner;
   int move;
   bool isInvalid;
   string moveString;
   srand((int) time(0));
   int corners[4] = {0,2,6,8};
   int sides[4] = {1,3,5,7};

   cout << "1|2|3\n-----\n4|5|6\n-----\n7|8|9\n\n";

   while (true) {
      //Player X decides what move they'll do.
      do {
         cout << "X: ";
         getline(cin, moveString);
         move = moveString[0] - '1';
         if (move > 8 || move < 0 || board[move] != 0) {
            cout << "Invalid input" << endl;
            isInvalid = true;
         } else {
            board[move] = 1;
            isInvalid = false;
            cout << endl;
         }
      } while (isInvalid);

      //Decides whether or not the game continues.
      if (gameOver(board) > 0) {
         outputBoard(board);
         break;
      }

      //Player O decides which move they'll do.
      bool good = false;
      for (int x = 2; x > 0; x--){
         possibleWinner = willWin(board, x);
         if (possibleWinner != -1) {
            board[possibleWinner] = 2;
            good = true;
            break;
         }
      }
      if (good);
      else if (board[4] == 0) board[4] = 2; //Middle.
      else if (exceptionalCase(board) > -1) board[exceptionalCase(board)] = 2; //Exception boards.
      else if (getSpace(board, corners) != -1) board[getSpace(board, corners)] = 2; //Corners
      else board[getSpace(board, sides)] = 2; //Sides

      //Prints the board to the screen.
      outputBoard(board);

      //Decides whether or not the game continues.
      if(gameOver(board)) break;

   }
   return 0;
}
```

Certainly! The problem describes a scenario where there are three boxes of tennis balls, each labeled incorrectly. One box contains only yellow balls, one contains only white balls, and one contains both yellow and white balls. The goal is to determine the correct labeling of the boxes based on observations and initial labels.

The symbols used in the propositional logic representation are as follows:

- O1Y: A yellow ball was observed from box 1.
- O2W: A white ball was observed from box 2.
- O3Y: A yellow ball was observed from box 3.
- L1W: Box 1 was initially labeled white.
- L2Y: Box 2 was initially labeled yellow.
- L3B: Box 3 was initially labeled with both colors.

The knowledge base includes constraints and implications related to the initial labeling, observations, and box contents. For instance, if a yellow ball is observed in a box, then that box contains yellow balls (O1Y → C1Y). Similarly, if a white ball is observed in a box, then that box contains white balls (O2W → C2W).

To prove that box 2 must contain white balls (C2W), we use the observation O2W → C2W. If a white ball is observed in box 2, then box 2 must indeed contain white balls.

In summary, the propositional logic knowledge base helps deduce the correct labeling of the boxes based on the given observations and initial labels. The conclusion that box 2 contains white balls is derived through logical implications and constraints defined in the knowledge base.