```
# Uninformed Search algorithms

def tree_search(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""
    frontier.append(Node(problem.initial))
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        frontier.extend(node.expand(problem))
    return None

def tree_search_count(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""
    expandidos=0
    frontier.append(Node(problem.initial))
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return (node,expandidos)
        expandidos+=1
        frontier.extend(node.expand(problem))
    return (None,expandidos)


def graph_search(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Figure 3.7]"""
    frontier.append(Node(problem.initial))
    explored = list()
    while frontier:
        node = frontier.pop()
        #print(problem.display(node.state))
        #print('--------------------\n\n')
        if problem.goal_test(node.state):
            return node
        zz = []
        #explored.append(node.state)
        explored = explored + [node.state]
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)
    return None

def graph_search_count(problem, frontier):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Figure 3.7]"""
    expandidos=0
    frontier.append(Node(problem.initial))
    explored = set()
```

```
    while frontier:
        node = frontier.pop()
        expandidos+=1
        if problem.goal_test(node.state):
            return (node,expandidos)
        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)
    return (None,expandidos)

def breadth_first_tree_search(problem):
    """Search the shallowest nodes in the search tree first."""
    return tree_search(problem, FIFOQueue())


def depth_first_tree_search(problem):
    """Search the deepest nodes in the search tree first."""
    return tree_search(problem, Stack())

def depth_first_tree_search_count(problem):
    """Search the deepest nodes in the search tree first."""
    return tree_search_count(problem, Stack())

def depth_first_graph_search(problem):
    """Search the deepest nodes in the search tree first."""
    return graph_search(problem, Stack())

def depth_first_graph_search_count(problem):
    """Search the deepest nodes in the search tree first."""
    return graph_search_count(problem, Stack())


def breadth_first_search(problem):
    """[Figure 3.11]"""
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node
    frontier = FIFOQueue()
    frontier.append(node)
    explored = set()
    while frontier:
        node = frontier.pop()
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    return child
                frontier.append(child)
    return None


def breadth_first_search_count(problem):
    """[Figure 3.11]"""
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return 0,node
```

```
    frontier = FIFOQueue()
    frontier.append(node)
    explored = set()
    while frontier:
        node = frontier.pop()
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    return len(explored),child
                frontier.append(child)
    return len(explored),None


def best_first_graph_search(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node
    frontier = PriorityQueue(min, f)
    frontier.append(node)
    explored = set()
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                incumbent = frontier[child]
                if f(child) < f(incumbent):
                    del frontier[incumbent]
                    frontier.append(child)
    return None

def best_first_graph_search_count(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return (node,len(explored))
    frontier = PriorityQueue(min, f)
```

```
    frontier.append(node)
    explored = set()
    while frontier:
        node = frontier.pop()
        #print('Testo se é objectivo, com custo',node.path_cost)
        #print(node.state)
        if problem.goal_test(node.state):
            return (node,len(explored))
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                #print('Sucessor com custo',child.path_cost)
                frontier.append(child)
            elif child in frontier:
                incumbent = frontier[child]
                if f(child) < f(incumbent):
                    del frontier[incumbent]
                    frontier.append(child)
    return (None,len(explored))

def uniform_cost_search(problem):
    """[Figure 3.14]"""
    return best_first_graph_search(problem, lambda node: node.path_cost)


def uniform_cost_search_count(problem):
    """[Figure 3.14]"""
    return best_first_graph_search_count(problem, lambda node: node.path_cost)

def best_first_graph_search_plus(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node
    frontier = PriorityQueue(min, f)
    frontier.append(node)
    explored = set()
    visited_not_explored={node.state}
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node
        explored.add(node.state)
        visited_not_explored.remove(node.state)
        for child in node.expand(problem):
            if child.state not in explored:
                if child.state not in visited_not_explored:
                    frontier.append(child)
                    visited_not_explored.add(child.state)
                else:
```

```
                    incumbent = frontier[child]
                    if f(child) < f(incumbent):
                        del frontier[incumbent]
                        frontier.append(child)
    return None

def uniform_cost_search_plus(problem):
    """[Figure 3.14]"""
    return best_first_graph_search_plus(problem, lambda node: node.path_cost)

def best_first_graph_search_plus_count(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node,0
    frontier = PriorityQueue(min, f)
    frontier.append(node)
    explored = set()
    visited_not_explored={node.state}
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node,len(explored)
        explored.add(node.state)
        visited_not_explored.remove(node.state)
        for child in node.expand(problem):
            if child.state not in explored:
                if child.state not in visited_not_explored:
                    frontier.append(child)
                    visited_not_explored.add(child.state)
                else:
                    incumbent = frontier[child]
                    if f(child) < f(incumbent):
                        del frontier[incumbent]
                        frontier.append(child)
    return None, len(explored)


def uniform_cost_search_plus_count(problem):
    """[Figure 3.14]"""
    return best_first_graph_search_plus_count(problem, lambda node: node.path_cost)

# em árvore o Best First e o A*

def best_first_tree_search_count(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
```

```
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""
    f = memoize(f, 'f')
    explored=0
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node,explored
    frontier = PriorityQueue(min, f)
    frontier.append(node)
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            return node,explored
        explored+=1
        frontier.extend(child for child in node.expand(problem))
    return None,explored

# ------------------------------------------------------------------

def depth_limited_search(problem, limit=50):
    """[Figure 3.17]"""
    def recursive_dls(node, problem, limit):
        if problem.goal_test(node.state):
            return node
        elif limit == 0:
            return 'cutoff'
        else:
            cutoff_occurred = False
            for child in node.expand(problem):
                result = recursive_dls(child, problem, limit - 1)
                if result == 'cutoff':
                    cutoff_occurred = True
                elif result is not None:
                    return result
            return 'cutoff' if cutoff_occurred else None

    # Body of depth_limited_search:
    return recursive_dls(Node(problem.initial), problem, limit)


def iterative_deepening_search(problem):
    """[Figure 3.18]"""
    for depth in range(sys.maxsize):
        result = depth_limited_search(problem, depth)
        if result != 'cutoff':
            return result
# ____________________________________________________________________________
# Bidirectional Search
# Pseudocode from https://webdocs.cs.ualberta.ca/%7Eholte/Publications/MM-AAAI2016.pdf

def bidirectional_search(problem):
    e = problem.find_min_edge()
    gF, gB = {problem.initial : 0}, {problem.goal : 0}
    openF, openB = [problem.initial], [problem.goal]
    closedF, closedB = [], []
    U = infinity
```

```
def extend(U, open_dir, open_other, g_dir, g_other, closed_dir):
    """Extend search in given direction"""
    n = find_key(C, open_dir, g_dir)

    open_dir.remove(n)
    closed_dir.append(n)

    for c in problem.actions(n):
        if c in open_dir or c in closed_dir:
            if g_dir[c] <= problem.path_cost(g_dir[n], n, None, c):
                continue

            open_dir.remove(c)

        g_dir[c] = problem.path_cost(g_dir[n], n, None, c)
        open_dir.append(c)

        if c in open_other:
            U = min(U, g_dir[c] + g_other[c])

    return U, open_dir, closed_dir, g_dir


def find_min(open_dir, g):
    """Finds minimum priority, g and f values in open_dir"""
    m, m_f = infinity, infinity
    for n in open_dir:
        f = g[n] + problem.h(n)
        pr = max(f, 2*g[n])
        m = min(m, pr)
        m_f = min(m_f, f)

    return m, m_f, min(g.values())


def find_key(pr_min, open_dir, g):
    """Finds key in open_dir with value equal to pr_min
    and minimum g value."""
    m = infinity
    state = -1
    for n in open_dir:
        pr = max(g[n] + problem.h(n), 2*g[n])
        if pr == pr_min:
            if g[n] < m:
                m = g[n]
                state = n

    return state


while openF and openB:
    pr_min_f, f_min_f, g_min_f = find_min(openF, gF)
    pr_min_b, f_min_b, g_min_b = find_min(openB, gB)
    C = min(pr_min_f, pr_min_b)

    if U <= max(C, f_min_f, f_min_b, g_min_f + g_min_b + e):
```

```
            return U

        if C == pr_min_f:
            # Extend forward
            U, openF, closedF, gF = extend(U, openF, openB, gF, gB, closedF)
        else:
            # Extend backward
            U, openB, closedB, gB = extend(U, openB, openF, gB, gF, closedB)

    return infinity

# ______________________________________________________________________________
# Informed (Heuristic) Search

greedy_best_first_graph_search = best_first_graph_search
# Greedy best-first search is accomplished by specifying f(n) = h(n).

def astar_search(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search(problem, lambda n: n.path_cost + h(n))

def astar_search_tree_count(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    return best_first_tree_search_count(problem, lambda n: n.path_cost + h(n))

def astar_search_plus_count(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search_plus_count(problem, lambda n: n.path_cost + h(n))
# ______________________________________________________________________________
# Other search algorithms

def recursive_best_first_search(problem, h=None):
    """[Figure 3.26]"""
    h = memoize(h or problem.h, 'h')

    def RBFS(problem, node, flimit):
        if problem.goal_test(node.state):
            return node, 0   # (The second value is immaterial)
        successors = node.expand(problem)
        if len(successors) == 0:
            return None, infinity
        for s in successors:
            s.f = max(s.path_cost + h(s), node.f)
        while True:
            # Order by lowest f value
            successors.sort(key=lambda x: x.f)
            best = successors[0]
            if best.f > flimit:
```

```
            return None, best.f
        if len(successors) > 1:
            alternative = successors[1].f
        else:
            alternative = infinity
        result, best.f = RBFS(problem, best, min(flimit, alternative))
        if result is not None:
            return result, best.f

    node = Node(problem.initial)
    node.f = h(node)
    result, bestf = RBFS(problem, node, infinity)
    return result

def hill_climbing(problem):
    """From the initial node, keep choosing the neighbor with highest value,
    stopping when no neighbor is better. [Figure 4.2]"""
    current = Node(problem.initial)
    while True:
        neighbors = current.expand(problem)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors,
                                     key=lambda node: problem.value(node.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current = neighbor
    return current.state

def exp_schedule(k=20, lam=0.005, limit=100):
    """One possible schedule function for simulated annealing"""
    return lambda t: (k * math.exp(-lam * t) if t < limit else 0)

def simulated_annealing(problem, schedule=exp_schedule()):
    """[Figure 4.5] CAUTION: This differs from the pseudocode as it
    returns a state instead of a Node."""
    current = Node(problem.initial)
    for t in range(sys.maxsize):
        T = schedule(t)
        if T == 0:
            return current.state
        neighbors = current.expand(problem)
        if not neighbors:
            return current.state
        next = random.choice(neighbors)
        delta_e = problem.value(next.state) - problem.value(current.state)
        if delta_e > 0 or probability(math.exp(delta_e / T)):
            current = next

def and_or_graph_search(problem):
    """[Figure 4.11]Used when the environment is nondeterministic and completely observable.
    Contains OR nodes where the agent is free to choose any action.
    After every action there is an AND node which contains all possible states
    the agent may reach due to stochastic nature of environment.
    The agent must be able to handle all possible states of the AND node (as it
    may end up in any of them).
    Returns a conditional plan to reach goal state,
```

```
    or failure if the former is not possible."""

    # functions used by and_or_search
    def or_search(state, problem, path):
        """returns a plan as a list of actions"""
        if problem.goal_test(state):
            return []
        if state in path:
            return None
        for action in problem.actions(state):
            plan = and_search(problem.result(state, action),
                              problem, path + [state, ])
            if plan is not None:
                return [action, plan]

    def and_search(states, problem, path):
        """Returns plan in form of dictionary where we take action plan[s] if we reach state s."""
        plan = {}
        for s in states:
            plan[s] = or_search(s, problem, path)
            if plan[s] is None:
                return None
        return plan

    # body of and or search
    return or_search(problem.initial, problem, [])


//---------------------------------------------------------------------

// Java program for implementation of Ford Fulkerson
// algorithm
import java.io.*;
import java.lang.*;
import java.util.*;
import java.util.LinkedList;

class MaxFlow {
    static final int V = 6; // Number of vertices in graph

    /* Returns true if there is a path from source 's' to
      sink 't' in residual graph. Also fills parent[] to
      store the path */
    boolean bfs(int rGraph[][], int s, int t, int parent[])
    {
        // Create a visited array and mark all vertices as
        // not visited
        boolean visited[] = new boolean[V];
        for (int i = 0; i < V; ++i)
            visited[i] = false;

        // Create a queue, enqueue source vertex and mark
        // source vertex as visited
        LinkedList<Integer> queue
            = new LinkedList<Integer>();
        queue.add(s);
        visited[s] = true;
        parent[s] = -1;
```

```
    // Standard BFS Loop
    while (queue.size() != 0) {
        int u = queue.poll();

        for (int v = 0; v < V; v++) {
            if (visited[v] == false
                && rGraph[u][v] > 0) {
                // If we find a connection to the sink
                // node, then there is no point in BFS
                // anymore We just have to set its parent
                // and can return true
                if (v == t) {
                    parent[v] = u;
                    return true;
                }
                queue.add(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    // We didn't reach sink in BFS starting from source,
    // so return false
    return false;
}

// Returns the maximum flow from s to t in the given
// graph
int fordFulkerson(int graph[][], int s, int t)
{
    int u, v;

    // Create a residual graph and fill the residual
    // graph with given capacities in the original graph
    // as residual capacities in residual graph

    // Residual graph where rGraph[i][j] indicates
    // residual capacity of edge from i to j (if there
    // is an edge. If rGraph[i][j] is 0, then there is
    // not)
    int rGraph[][] = new int[V][V];

    for (u = 0; u < V; u++)
        for (v = 0; v < V; v++)
            rGraph[u][v] = graph[u][v];

    // This array is filled by BFS and to store path
    int parent[] = new int[V];

    int max_flow = 0; // There is no flow initially

    // Augment the flow while there is path from source
    // to sink
    while (bfs(rGraph, s, t, parent)) {
        // Find minimum residual capacity of the edges
```

```
            // along the path filled by BFS. Or we can say
            // find the maximum flow through the path found.
            int path_flow = Integer.MAX_VALUE;
            for (v = t; v != s; v = parent[v]) {
                u = parent[v];
                path_flow
                    = Math.min(path_flow, rGraph[u][v]);
            }

            // update residual capacities of the edges and
            // reverse edges along the path
            for (v = t; v != s; v = parent[v]) {
                u = parent[v];
                rGraph[u][v] -= path_flow;
                rGraph[v][u] += path_flow;
            }

            // Add path flow to overall flow
            max_flow += path_flow;
        }

        // Return the overall flow
        return max_flow;
    }

    // Driver program to test above functions
    public static void main(String[] args)
        throws java.lang.Exception
    {
        // Let us create a graph shown in the above example
        int graph[][] = new int[][] {
            { 0, 16, 13, 0, 0, 0 }, { 0, 0, 10, 12, 0, 0 },
            { 0, 4, 0, 0, 14, 0 },  { 0, 0, 9, 0, 0, 20 },
            { 0, 0, 0, 7, 0, 4 },   { 0, 0, 0, 0, 0, 0 }
        };
        MaxFlow m = new MaxFlow();

        System.out.println("The maximum possible flow is "
                           + m.fordFulkerson(graph, 0, 5));
    }
}

//---------------------------------------------------------------------------
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Queue;

class Graph {

    public static final boolean DIRECT = true,
            UNDIRECT = !DIRECT;

    public static final boolean SPARSE = true,
            NOTSPARSE = !SPARSE;

    private boolean isSparse; // is the graph using a sparse representation?
```

```
private boolean isDirected; // is the graph directed?
private int size; // number of nodes

// Non-sparse representation:
// A 2D array where rows and columns represent the nodes and
// each position represents the weight between nodes (zero means no connection)
// The graph will consist of V nodes and E edges (E <= V^2)

private int[][] graphMatrix;

// Sparse representation
// An array of hashmaps to represent sparse graphs
// The edge (i,j,w) will be added as graphList[i].put(j,w)

private ArrayList<HashMap<Integer, Integer>> graphList;

////////////////////////// BASIC METHODS ////////////////////////////////////

// by default we use a matrix to represent a graph, ie, a non-sparse
// representation
public Graph(int nodes, boolean graphType) {
    size = nodes;
    isDirected = graphType;
    isSparse = false;
    graphMatrix = new int[size][size];
}

public Graph(int nodes, boolean graphType, boolean sparse) {
    size = nodes;
    isDirected = graphType;
    isSparse = sparse;

    if (isSparse) {
        graphList = new ArrayList<HashMap<Integer, Integer>>(size);
        for (int i = 0; i < size; i++)
            graphList.add(i, new HashMap<Integer, Integer>());
    } else
        graphMatrix = new int[size][size];
}

/**
 * Add edge to graph
 */
public void add(int from, int to, int weight) {
    if (isSparse) {
        graphList.get(from).put(to, weight);
        if (isDirected == UNDIRECT)
            graphList.get(to).put(from, weight);
    } else {
        graphMatrix[from][to] = weight;
        if (isDirected == UNDIRECT)
            graphMatrix[to][from] = weight;
    }
}

public void add(int from, int to) {
    add(from, to, 1);
```

```
}

/**
 * Remove edge to graph
 */
public void remove(int from, int to) {
    if (isSparse) {
        graphList.get(from).remove(to);
        if (isDirected == UNDIRECT)
            graphList.get(to).remove(from);
    } else
        add(from, to, 0); // remove edge
}

public int weight(int from, int to) {
    if (isSparse) {
        Integer w = graphList.get(from).get(to);
        return w == null ? 0 : w;
    } else
        return graphMatrix[from][to];
}

public int size() {
    return size;
}

/**
 * Remove all in-edges and out-edges from/into node
 */
public void isolate(int node) {
    if (isSparse) {
        graphList.set(node, new HashMap<Integer, Integer>()); // remove out-edges
        for (int i = 0; i < size; i++)
            graphList.get(i).remove(node); // remove in-edges (slow)
    } else {
        if (isDirected == DIRECT)
            graphMatrix[node] = new int[size];
        for (int i = 0; i < size; i++)
            remove(i, node);
    }
}

/**
 * @param node The node which successors we need
 * @requires a directed graph
 * @return an array with the indexes of the node's successors
 */
public int[] sucessors(int node) {
    ArrayList<Integer> l = new ArrayList<Integer>();

    if (isSparse) {
        for (Integer successor : graphList.get(node).keySet())
            l.add(successor);
    } else {
        for (int i = 0; i < size; i++)
            if (weight(node, i) != 0)
                l.add(i);
```

```
    }

    return list2array(l);
}

/**
 * @param node The node which predecessors we need
 * @requires a directed graph
 * @return an array with the indexes of the node's predecessors
 */
public int[] predecessors(int node) {
    ArrayList<Integer> l = new ArrayList<Integer>();

    if (isSparse) {
        for (int i = 0; i < size; i++) { // slow
            Integer weight = graphList.get(i).get(node);
            if (weight != null)
                l.add(i);
        }
    } else {
        for (int i = 0; i < size; i++)
            if (weight(i, node) != 0)
                l.add(i);
    }

    return list2array(l);
}

/**
 * Make a copy of this
 *
 * @return the reference to the copy
 */
@SuppressWarnings("unchecked")
public Graph copy() {

    Graph cp = new Graph(this.size, this.isDirected, this.isSparse);

    if (isSparse) {
        for (int i = 0; i < cp.size; i++)
            cp.graphList.set(i,
                    (HashMap<Integer, Integer>) this.graphList.get(i).clone());
    } else {
        for (int i = 0; i < cp.size; i++)
            cp.graphMatrix[i] = this.graphMatrix[i].clone();
    }

    return cp;
}

public String toString() {
    StringBuffer sb = new StringBuffer();

    if (isSparse) {
        for (int i = 0; i < size; i++)
            for (Integer j : graphList.get(i).keySet()) {
                int weight = graphList.get(i).get(j);
```

```
                if (weight != 0)
                    if (weight == 1)
                        sb.append(i + "->" + j + " "); // don't show weights 1
                    else
                        sb.append(i + "-{" + weight + "}->" + j + " ");
            }
    } else {
        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                if (weight(i, j) != 0)
                    if (weight(i, j) == 1)
                        sb.append(i + "->" + j + " "); // don't show weights 1
                    else
                        sb.append(i + "-{" + weight(i, j) + "}->" + j + " ");
    }

    return sb.toString();
}

private int[] list2array(ArrayList<Integer> list) {
    int[] array = new int[list.size()];
    int index = 0;

    for (int elem : list)
        array[index++] = elem;
    return array;
}

public boolean isConnected(int from, int to) {
    ArrayList<Integer> visitedNodes = new ArrayList<>();
    Queue<Integer> queuePorExpandir = new ArrayDeque<>(); //fronteira //largura

    queuePorExpandir.add(from);
    while (!queuePorExpandir.isEmpty()) {
        int next = queuePorExpandir.poll();
        if (next == to)
            return true;
        visitedNodes.add(next);
        int[] neighboors = sucessors(next);
        for (int j = 0; j < neighboors.length; j++)
            if (!visitedNodes.contains(neighboors[j]))
                queuePorExpandir.add(neighboors[j]);
                // adicionamos os nos ao final da fila
    }
    return false;
}



 // -----------------FLOOD FILL--------------------------------
static int dr[] = { 1, 1, 0, -1, -1, -1, 0, 1 }; // trick to explore an implicit 2D grid
static int dc[] = { 0, 1, 1, 1, 0, -1, -1, -1 }; // S,SE,E,NE,N,NW,W,SW neighbors

// static int dr[] = {1,0,-1, 0}; //
// static int dc[] = {0,1, 0,-1}; // S,E,N,W neighbors

/**
```

```
     * Use graph structure to simulate a 2D grid
     * Replaces the old color by the new considering all directions,
     * and returns the total number of replacements
     *
     * @ensures a total chaotic graph if interpreted by default
     * @complexity O(V + E)
     * @throws Exception If representation is non-sparce (ie, it needs the adjacency matrix)
     */
    public int floodFill(int row, int col, int oldColor, int newColor) throws Exception {

        if (isSparse)
            throw new Exception("Flood Fill only works on a non-sparce representation");

        if (row < 0 || row >= size || col < 0 || col >= size)
            return 0; // outside grid

        if (graphMatrix[row][col] != oldColor)
            return 0; // does not have old color

        graphMatrix[row][col] = newColor; // recolors vertex to avoid cycling
        int ans = 1; // adds 1 because vertex (row, col) was replaced
        for (int d = 0; d < 8; d++)
            ans += floodFill(row + dr[d], col + dc[d], oldColor, newColor);
        return ans;
    }
}
//----------------------------Prob D 2022-----------------------------------------------------------

public class ProblemD {

    private static final String[] WEEKS = {"Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
    private static final List<String> MONTHS = Arrays.asList("Jan", "Feb", "Mar",
        "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Nov", "Dec");
    private static final Set<String> MONTHS_30 = new HashSet<>(Arrays.asList("Sep", "Apr",
        "Jun", "Nov"));

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String date = sc.nextLine();
            String[] split = date.split(" ");
            int day = Integer.parseInt(split[0]);
            String month = split[1];
            int year = Integer.parseInt(split[2]);

            int weekDay = 0;
            int yearDiff = year - 1900;
            int bix = 0;
            for (int i = 1900; i < year; i++) {
                if ((i % 4 == 0 && i % 100 != 0) || (i % 4 == 0 && i % 100 == 0 && i % 400 == 0)) {
                    bix++;
                }
            }
            weekDay = (weekDay + yearDiff * 365 + bix) % 7;

            int monthIndex = MONTHS.indexOf(month);
```

```
            for (int i = 0; i < monthIndex; i++) {
                String m = MONTHS.get(i);
                if (MONTHS_30.contains(m))
                    weekDay = (weekDay + 30) % 7;
                else if (m.equals("Feb")) {
                    int days = 28;
                    if ((year % 4 == 0 && year % 100 != 0) || (year % 4 == 0 &&
                        year % 100 == 0 && year % 400 == 0))
                        days++;
                    weekDay = (weekDay + days) % 7;
                } else
                    weekDay = (weekDay + 31) % 7;
            }

            weekDay = (weekDay + day - 1) % 7;
            System.out.println(WEEKS[weekDay]);
        }
    }
}
//---------------------------Prob E 2022-------------------------------------------------------------------
public class ProblemE {

    public static class Pos {
        int x;
        int y;

        public Pos(int x, int y) {
            this.x = x;
            this.y = y;
        }

        public boolean equals(Object other) {
            return other instanceof Pos pos && pos.x == this.x && pos.y == this.y;
        }
    }

    public static int dist(Pos pos1, Pos pos2) {
        return Math.abs(pos1.x - pos2.x) + Math.abs(pos1.y - pos2.y);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int tests = sc.nextInt();
        for (int i = 0; i < tests; i++) {
            int n = sc.nextInt();
            int m = sc.nextInt();
            int[] ns = new int[n];
            int[] ms = new int[m];
            for (int j = 0; j < n; j++) {
                ns[j] = sc.nextInt();
            }
            for (int j = 0; j < m; j++) {
                ms[j] = sc.nextInt();
            }
            Pos ana = new Pos(ns[0], ms[0]);
            Pos pedro = new Pos(ns[n - 1], ms[m - 1]);
            Pos[] allPos = new Pos[n * m];
```

```
            int index = 0;
            for (int nn : ns) {
                for (int mm : ms) {
                    allPos[index++] = new Pos(nn, mm);
                }
            }
            Pos minPos = ana;
            int dist = dist(ana, pedro);
            for (Pos pos : allPos) {
                int distPos = Math.abs(dist(pos, ana) - dist(pos, pedro));
                if (distPos < dist) {
                    dist = distPos;
                    minPos = pos;
                }
            }
            System.out.println(minPos.x + " " + minPos.y);
        }
    }
}
}
//---------------------------Prob E 2022-------------------------------------------------------------------
public class ProblemF {

    private static final char[] ACTIONS = {'u', 'd', 'e', 'w'};

    private static class Node {
        public int x, y;
        public int cost;

        public Node(int x, int y, int cost) {
            this.x = x;
            this.y = y;
            this.cost = cost;
        }

        public boolean equals(Object other) {
            return other instanceof Node n && n.x == this.x && n.y == this.y;
        }

        public int hashCode() {
            return Objects.hash(this.x, this.y);
        }
    }

    public static List<Character> actions(Node current, String[] map) {
        List<Character> actions = new ArrayList<>();
        for (char a : ACTIONS) {
            if (movePos(current, a, map) != null)
                actions.add(a);
        }
        return actions;
    }

    public static Node movePos(Node current, char dir, String[] map) {
        int xMove = dir == 'e' ? 1 : dir == 'w' ? -1 : 0;
        int yMove = dir == 'd' ? 1 : dir == 'u' ? -1 : 0;
        int x = current.x;
```

```
    int y = current.y;
    while (x >= 0 && y >= 0 && x < map[0].length() && y < map.length) {
        if (map[y].charAt(x) == 'H')
            return new Node(x, y, current.cost + 1);
        if (map[y].charAt(x) == 'O')
            return new Node(x - xMove, y - yMove, current.cost + 1);
        x += xMove;
        y += yMove;
    }
    return null;
}

public static Node[] expandNode(Node node, String[] map) {
    List<Character> actions = actions(node, map);
    Node[] result = new Node[actions.size()];
    for (int i = 0; i < actions.size(); i++) {
        result[i] = movePos(node, actions.get(i), map);
    }
    return result;
}

public static int search(Node root, String[] map) {
    Set<Node> explored = new HashSet<>();
    Queue<Node> queue = new ArrayDeque<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        Node node = queue.poll();
        // System.out.println(actions(root, map));
        explored.add(node);
        Node[] childs = expandNode(node, map);
        for (Node child : childs) {
            if (map[child.y].charAt(child.x) == 'H')
                return child.cost;
            if (!explored.contains(child))
                queue.add(child);
        }
    }
    return -1;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int rows = sc.nextInt();
    int columns = sc.nextInt();
    int numTests = sc.nextInt();
    sc.nextLine();
    String[] map = new String[rows];
    for (int i = 0; i < rows; i++) {
        map[i] = sc.nextLine();
    }

    for (int i = 0; i < numTests; i++) {
        int y = sc.nextInt();
        int x = sc.nextInt();
        Node root = new Node(x - 1, y - 1, 0);
        int sol = search(root, map);
        if (sol == -1)
```

```
                System.out.println("Stuck");
            else
                System.out.println(sol);
        }

    }
}
//---------------------------------------Prob C-------------------------------------------
public class ProblemC {

    public static class Node {
        String piece;
        String postion;
        int cost;

        public Node(String piece, String position, int cost) {
            this.piece = piece;
            this.postion = position;
            this.cost = cost;
        }
    }

    public static Set<String> getRookPositions(String position) {
        Set<String> positions = new HashSet<>();
        for (int i = 1; i <= 8; i++) {
                positions.add(position.charAt(0) + "" + i );
        }
        for (int i = 'A'; i <= 'H'; i++) {
            positions.add((char) i + "" + position.charAt(1));
        }
        return positions;
    }

    public static Set<String> getBishopPositions(String position) {
        Set<String> positions = new HashSet<>();
        return positions;
    }

    public static Set<String> getKnightPositions(String position) {
        Set<String> positions = new HashSet<>();

        return positions;
    }

    public static Set<String> getPositions(String piece, String position) {
        Set<String> positions = new HashSet<>();
        if (piece.equals("Q") || piece.equals("C"))
            positions.addAll(getRookPositions(position));
        if (piece.equals("C") || piece.equals("A"))
            positions.addAll(getKnightPositions(position));
        if (piece.equals("A") || piece.equals("Q"))
            positions.addAll(getBishopPositions(position));
        return positions;
    }

    public static String getSquareColor(String position) {
        int column = position.charAt(0) - 'A';
```

```
        int row = position.charAt(1) - '0';
        if ((row % 2 == 0 && column % 2 == 0) || (row % 2 == 1 && column % 2 == 1))
            return "black";
        return "white";
    }

    public static int cost(String piece, String position, String end) {
        Set<String> positions = getPositions(piece, position);
        if (positions.contains(end))
            return 1;
        if (piece.equals("Q") || piece.equals("C"))
            return 2;
        if (getSquareColor(position).equals(getSquareColor(end)))
            return 2;
        return 3;
    }

    public static void main(String[] args) {
        System.out.println(getRookPositions("A1"));
        System.out.println(getKnightPositions("D4"));
        // Scanner sc = new Scanner(System.in);
        // int nTests = sc.nextInt();
        // for (int i = 0; i < nTests; i++) {
        //     int nPlaces = sc.nextInt();
        //     String start = sc.next();
        //     Node startA = new Node("A", start, 0);
        //     Node startC = new Node("C", start, 0);
        //     Node startQ = new Node("Q", start, 0);
        //     sc.nextLine();
        //     for (int j = 0; j < nPlaces; j++) {
        //         String place = sc.nextLine();
        //     }
        // }
    }
}

//-----------------------------------prob A-----------------------------------------------------------
public class ProblemA {

    private static class Applicant {
        public String name;
        public int followers, cv;

        public Applicant(String name, int follow, int cv) {
            this.name = name;
            this.followers = follow;
            this.cv = cv;
        }

        @Override
        public boolean equals(Object other) {
            return other instanceof Applicant app && app.name.equals(name);
        }
    }

    private static class Node {
        Applicant applicant;
```

```
    Node parent;

    public Node(Node parent, Applicant app) {
        this.parent = parent;
        this.applicant = app;
    }
}

public static int calcCV(Node node) {
    int cv = 0;
    while (node != null) {
        cv += node.applicant.cv;
        node = node.parent;
    }
    return cv;
}

public static int calcFollowers(Node node) {
    int f = 0;
    while (node != null) {
        f += node.applicant.followers;
        node = node.parent;
    }
    return f;
}

public static Set<Applicant> getApplicants(Node node) {
    Set<Applicant> apps = new HashSet<>();
    while (node != null) {
        apps.add(node.applicant);
        node = node.parent;
    }
    return apps;
}

public static Node depthSearch(Node root, List<Applicant> app, int maxFollowers) {
    Node sol = null;
    Deque<Node> nodes = new ArrayDeque<>();
    nodes.add(root);
    System.out.println("STARTING FROM " + root.applicant.name);
    while (!nodes.isEmpty()) {
        Node node = nodes.pop();
        // System.out.println("POP " + node.applicant.name);
        // System.out.println(node);
        Set<Applicant> apps = getApplicants(node);
        int followers = calcFollowers(node);
        List<Node> childs = app.stream()
            .map(a -> new Node(node, a))
            .filter(n -> !apps.contains(n.applicant))
            .filter(n -> followers + n.applicant.followers <= maxFollowers)
            .toList();
        if (childs.isEmpty()) {
            if (sol == null)
                sol = node;
            else if (calcCV(node) > calcCV(sol))
                sol = node;
        } else {
```

```
                nodes.addAll(childs);
            }
        }
        return sol;
    }

    public static Node searchSol(List<Applicant> apps, int maxFollowers) {
        Node sol = null;
        for (Applicant a : apps) {
            Node res = depthSearch(new Node(null, a), apps, maxFollowers);
            if (sol == null)
                sol = res;
            if (calcCV(sol) < calcCV(res))
                sol = res;
        }
        return sol;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int capacity = sc.nextInt();
        int n = sc.nextInt();
        sc.nextLine();
        List<Applicant> apps = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            String name = sc.next();
            int followers = sc.nextInt();
            int cv = sc.nextInt();
            Applicant app = new Applicant(name, followers, cv);
            apps.add(app);
            if (i != n - 1)
                sc.nextLine();
        }
        Node sol = searchSol(apps, capacity);
        Set<Applicant> res = getApplicants(sol);
        System.out.println(res.size() + " " + calcFollowers(sol) + " " + calcCV(sol));
        res.stream().forEach(a -> System.out.println(a.name));
    }
}
//----------------------weekly day-------------------------
import java.util.Calendar
Calendar c = Calendar.getInstance();
c.setTime(yourDate);
int dayOfWeek = c.get(Calendar.DAY_OF_WEEK);

//------------------------------------------------------------------------
import java.util.Date
SimpleDateFormat formatter = new SimpleDateFormat("dd-MMM-yyyy", Locale.ENGLISH);
String dateInString = "7-Jun-2013";
Date date = formatter.parse(dateInString);
```