CSCI 36200: Data Structures  
Programming Assignment 1  
Instructor: Dr. Snehasis Mukhopadhyay

Due date: February 15, 2018

Contents

[Description 1](#_Toc505793469)

[How to build and Run 5](#_Toc505793470)

[Source code 5](#_Toc505793471)

[Output 21](#_Toc505793472)

[Run 1 21](#_Toc505793473)

[Run 2 23](#_Toc505793474)

[Run 3 25](#_Toc505793475)

# Description

The program uses recursive procedure to solve the Kight’s Tour problem.

The first 32 moves will be implemented by Warnsdorff’s Algorithm.

The remaining moves is implemented by using backtracking with Stack.

The program implements SinglyLinkedList class with common methods: size, isEmpty, get, removeFirst, add…The singly linked list contains the nodes. Each node stores Object data and a reference to next node.

head

NODE

NODE

NODE

The program also the Stack class. The Stack class uses SinglyLinkedList as its internal data structure. The head node is used as the top of stack. The push operation add new node as head node. The pop operation remove the head node. The methods of Stack call the methods of SinglyLinkedList class.

Stack

- data: SinglyLinkedList

SinglyLinkedList

Warnsdorff algorithm

Input: initial position, empty chess board

Output: chess board is loaded with number of steps N: 1, 2...N

Pseudo code

P = initial position

Set Moves = 1

Set location of P in chess board = 1

Moves = Moves + 1

For N times Do

Calculate S is the accessible positions from P (neighbors based on knight move)

P = minimum position from S (based on number of positions that can moved from s in S)

Set location of P in chess board = 1

Moves = Moves + 1

End for

Backtracking with Stack algorithm

This algorithm uses a stack to try the knight then moves its accessible positions from current positions to stack. If the solution is not found, the knight is removed from that position and the next position from stack is checked next (backtracking).

The algorithm uses Stack. It does not use recursive.

Pseudo code

Set current = last position after Warnsdorff algorithm

Set Moves = 24

Create Stack

Add all the accessible positions from P (neighbors based on knight move) to Stack

While stack is not empty and chess board is full (loaded from 1 to 64)

current = stack.pop()

If chessboard[current] is not empty then

Set current position is -1

Moves = Moves - 1

End if

Set current position is Moves

Moves = Moves + 1

If board is full then

Break

End if

S = neighbors of current

If S is empty then

Set current position is -1

Moves = Moves - 1

Else

stack.push(current)

add all the accessible positions from P (neighbors based on knight move) to Stack

End if

End while

If board is full then

Print board

Else

Print solution not found

End if

Class diagram

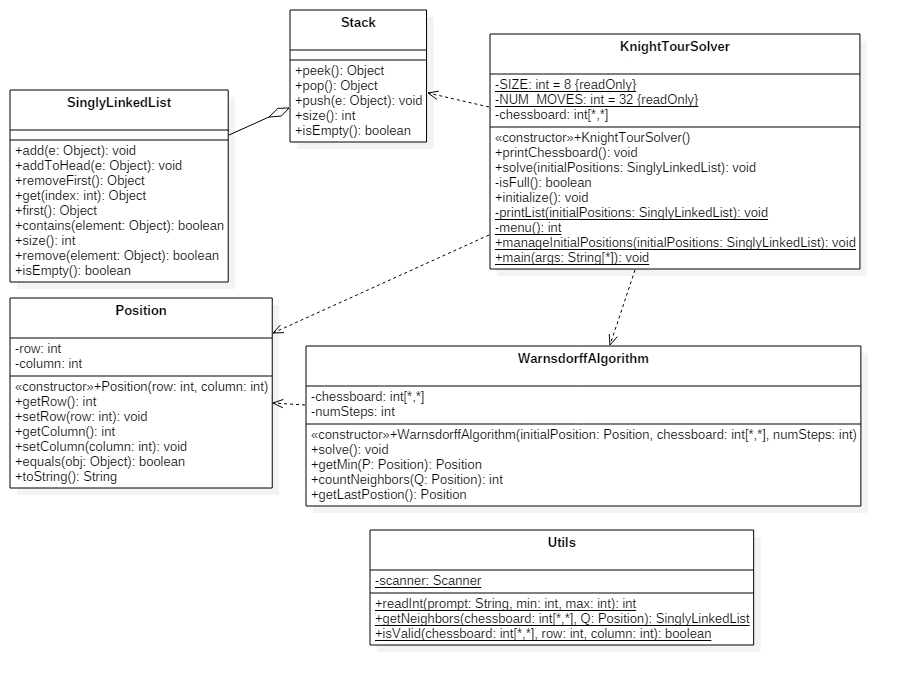


Figure Class diagram

SinglyLinkedList represents a singly link list

Stack represents a stack. It uses SinglyLinkedList object as internal data structure

Warnsdorff algorithm implements the problem in first 32 steps

KnightTourSolver contains main method that read the list of initial posisitions then allows users to use a menu to add, edit, and remove the position. Finally, that locations are used as initial positions in Knight Tour problem.

# How to build and Run

Open terminal or window command and enter

javac \*.java

Then run by the following command:

java KnightTourSolver

# Source code

KnightTourSolver.java

/\*\*

\* CSCI 36200: Data Structures

\* Programming Assignment 1

\* Instructor: Dr. Snehasis Mukhopadhyay

\*

\* Due date: February 15, 2018

\*

\* The program uses Warnsdorff Algorithm for 32 steps

\* then uses backtracking with stack to solve the remaining steps

\*

\*

\*/

**public** **class** KnightTourSolver {

/\*\*

\* 8 x 8 chess board

\*/

**private** **static** **final** **int** ***SIZE*** = 8;

/\*\*

\* apply Warndoff's heuristic for the first 32 moves

\*/

**private** **static** **final** **int** ***NUM\_MOVES*** = 32;

/\*\*

\* chess board

\*/

**private** **int**[][] chessboard = **new** **int**[***SIZE***][***SIZE***];

/\*\*

\* constructor

\*/

**public** KnightTourSolver(){

initialize();

}

/\*\*

\* print chess board

\*/

**public** **void** printChessboard(){

**for** (**int** r = 0; r < ***SIZE***; r++){

**for** (**int** c = 0; c < ***SIZE***; c++){

System.***out***.printf("%4d", chessboard[r][c]);

}

System.***out***.println();

}

System.***out***.println();

}

/\*\*

\* solve the problem

\* by moving by Warnsdorff Algorithm

\* and use backtracking (Stack)

\*/

**public** **void** solve(SinglyLinkedList initialPositions){

//get the positions from list and solve one by one

Position initialPosition = (Position)initialPositions.removeFirst();

**while** (initialPosition != **null**){

WarnsdorffAlgorithm alg = **new** WarnsdorffAlgorithm(initialPosition, chessboard, ***NUM\_MOVES***);

alg.solve();

//use backtracking for remaining steps

Position current = alg.getLastPostion();

**int** moves = ***NUM\_MOVES*** + 1;

Stack stack = **new** Stack();

//add neighbors to stack

SinglyLinkedList neighbors = Utils.*getNeighbors*(chessboard, current);

**while** (!neighbors.isEmpty()){

Position pos = (Position)neighbors.removeFirst();

stack.push(pos);

}

//run the algorithm

**while** (!stack.isEmpty() && !isFull()){

current = (Position)stack.pop();

//do not check again, all paths have failed

**if** (chessboard[current.getRow()][current.getColumn()] != -1){

chessboard[current.getRow()][current.getColumn()] = -1;

moves--;

**continue**;

}

chessboard[current.getRow()][current.getColumn()] = moves;

moves++;

//printChessboard();

//solution found?

**if** (isFull()){

**break**;

}

//get its neighbors

neighbors = Utils.*getNeighbors*(chessboard, current);

**if** (neighbors.size() == 0){ //can not move more

chessboard[current.getRow()][current.getColumn()] = -1;

moves--;

}**else**{

stack.push(current);

//try with next cell

**while** (!neighbors.isEmpty()){

Position pos = (Position)neighbors.removeFirst();

stack.push(pos);

}

}

}

//print initial position

System.***out***.printf("Initial position: row = %d, column = %d\n",

initialPosition.getRow() + 1, initialPosition.getColumn() + 1);

**if** (isFull()){

//print chess board

printChessboard();

}**else**{

System.***out***.println("Solution not found!");

}

//next initial position

initialPosition = (Position)initialPositions.removeFirst();

//clear chess board

initialize();

}

}

/\*\*

\* check if chess board is loaded with all knight moves

\* **@return**

\*/

**private** **boolean** isFull(){

//iterate each cells and check empty

**for** (**int** r = 0; r < ***SIZE***; r++){

**for** (**int** c = 0; c < ***SIZE***; c++){

**if** (chessboard[r][c] == -1){

**return** **false**;

}

}

}

**return** **true**;

}

/\*\*

\* initialize the chess board with -1

\*/

**public** **void** initialize(){

**for** (**int** r = 0; r < ***SIZE***; r++){

**for** (**int** c = 0; c < ***SIZE***; c++){

chessboard[r][c] = -1;

}

}

}

/\*\*

\* print list

\* **@param** initialPositions

\*/

**private** **static** **void** printList(SinglyLinkedList initialPositions){

**if** (initialPositions.size() == 0){

System.***out***.println("The initial positions list is empty");

System.***out***.println("");

**return**;

}

System.***out***.println("Current initial positions: ");

//print list

**int** size = initialPositions.size();

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

Position pos = (Position)initialPositions.get(i);

System.***out***.printf("%d. (row = %d, column = %d)\n", i + 1,

pos.getRow() + 1, pos.getColumn() + 1);

}

System.***out***.println("");

}

/\*\*

\* display menu

\*/

**private** **static** **int** menu(){

System.***out***.println("1. add");

System.***out***.println("2. edit");

System.***out***.println("3. remove");

System.***out***.println("4. exit");

**return** Utils.*readInt*("Please enter your selection? ", 1, 4);

}

/\*\*

\* add, delete, or modify the list of initial positions

\*

\* **@param** initialPositions initial positions

\*/

**public** **static** **void** manageInitialPositions(SinglyLinkedList initialPositions){

//position information

**int** row, column;

Position pos;

Position newPos; //modified position

**int** index; //chosen index of position

//print list

*printList*(initialPositions);

//menu

**int** selection = *menu*();

**while** (selection != 4){

**switch**(selection){

**case** 1: //add

row = Utils.*readInt*("Please enter row index [1-8]. Enter 0 to exit: ", 0, 8);

**if** (row == 0){

**break**;

}

column = Utils.*readInt*("Please enter column index [1-8]. Enter 0 to exit: ", 0, 8);

**if** (column == 0){

**break**;

}

pos = **new** Position(row - 1, column - 1);

**if** (initialPositions.contains(pos)){

System.***out***.println("You entered that position already");

}**else**{

initialPositions.add(pos);

}

**break**;

**case** 2://edit

**if** (initialPositions.size() == 0){

System.***out***.println("There is no initial position. Please add the initial position");

}**else**{

//print list

*printList*(initialPositions);

index = Utils.*readInt*("Please enter a position: ", 1, initialPositions.size());

pos = (Position)initialPositions.get(index - 1);

row = Utils.*readInt*("Please enter row index [1-8]. Enter 0 to exit: ", 0, 8);

**if** (row == 0){

**break**;

}

column = Utils.*readInt*("Please enter column index [1-8]. Enter 0 to exit: ", 0, 8);

**if** (column == 0){

**break**;

}

//different row, column

**if** (pos.getRow() != row - 1 || pos.getColumn() != column - 1){

newPos = **new** Position(row - 1, column - 1);

**if** (initialPositions.contains(newPos)){

System.***out***.println("You entered that position already");

}**else**{

pos.setRow(row - 1);

pos.setColumn(column - 1);

}

}

}

**break**;

**case** 3: //remove

//print list

*printList*(initialPositions);

index = Utils.*readInt*("Please enter a position", 1, initialPositions.size());

pos = (Position)initialPositions.get(index - 1);

initialPositions.remove(pos);

**break**;

}

//print list

*printList*(initialPositions);

//next selection

selection = *menu*();

}

}

/\*\*

\* main method

\* **@param** args not used

\*/

**public** **static** **void** main(String[] args) {

//create KnightTourSolver

KnightTourSolver solver = **new** KnightTourSolver();

//list of initial positions

SinglyLinkedList initialPositions = **new** SinglyLinkedList();

//read initial positions

**while** (**true**){

**int** row = Utils.*readInt*("Please enter row index [1-8]. Enter 0 to exit: ", 0, 8);

**if** (row == 0){

**break**;

}

**int** column = Utils.*readInt*("Please enter column index [1-8]. Enter 0 to exit: ", 0, 8);

**if** (column == 0){

**break**;

}

Position pos = **new** Position(row - 1, column - 1);

**if** (initialPositions.contains(pos)){

System.***out***.println("You entered that position already");

}**else**{

initialPositions.add(pos);

}

}//end read positions

System.***out***.println();

//edit list

*manageInitialPositions*(initialPositions);

**if** (initialPositions.size() == 0){

System.***out***.println("There is no initial position");

}**else**{

//solve the problem

solver.solve(initialPositions);

}

}

}

Position.java

/\*\*

\* Position represent a position

\*/

**public** **class** Position{

/\*\*

\* row

\*/

**private** **int** row;

/\*\*

\* column

\*/

**private** **int** column;

/\*\*

\* constructor

\* **@param** row row

\* **@param** column column

\*/

**public** Position(**int** row, **int** column) {

**this**.row = row;

**this**.column = column;

}

/\*\*

\* getter of row

\* **@return** the row

\*/

**public** **int** getRow() {

**return** row;

}

/\*\*

\* setter of row

\* **@param** row the row to set

\*/

**public** **void** setRow(**int** row) {

**this**.row = row;

}

/\*\*

\* getter of column

\* **@return** the column

\*/

**public** **int** getColumn() {

**return** column;

}

/\*\*

\* setter of column

\* **@param** column the column to set

\*/

**public** **void** setColumn(**int** column) {

**this**.column = column;

}

//compare two positions by (row, column) pairs

@Override

**public** **boolean** equals(Object obj) {

Position other = (Position) obj;

**return** (column == other.column && row == other.row);

}

@Override

**public** String toString() {

**return** row + ", " + column;

}

}

SinglyLinkedList.java

/\*\*

\* The class implements the singly link list

\* that contains nodes

\* Each node has data and the reference to next node

\* (or null)

\*

\*/

**public** **class** SinglyLinkedList {

/\*\*

\* node in linked list

\*/

**private** **class** Node{

/\*\*

\* data of node

\*/

Object data;

/\*\*

\* reference to next node

\*/

Node next;

/\*\*

\* constructor

\* **@param** data an element

\*/

**public** Node(Object data) {

**this**.data = data;

}

}

/\*\*

\* head node of singly linked list

\*/

**private** Node head;

/\*\*

\* add an element to end of list

\* **@param** e new element

\*/

**public** **void** add(Object e){

**if** (head == **null**){

head = **new** Node(e);

}**else**{

Node current = head;

//iterate the list

**while** (current.next != **null**){

current = current.next;

}

//insert as tail node

current.next = **new** Node(e);

}

}

/\*\*

\* add an element to head of list

\* **@param** e new element

\*/

**public** **void** addToHead(Object e){

**if** (head == **null**){

head = **new** Node(e);

}**else**{

Node newNode = **new** Node(e);

newNode.next = head;

head = newNode;

}

}

/\*\*

\* remove the first element in the linked list.

\* **@return** first element or null if list is empty

\*/

**public** Object removeFirst() {

**if** (head == **null**) {

**return** **null**;

} **else** {

Object element = head.data;

//move head to next node

head = head.next;

**return** element;

}

}

/\*\*

\* get element at index, return null if index is invalid.

\* **@return** element or null if index is invalid

\*/

**public** Object get(**int** index) {

**if** (index < 0 || index >= size()){

**return** **null**;

}

Node current = head;

//iterate the list to get object at index

**for** (**int** i = 0; i < index; i++){

current = current.next;

}

**return** current.data;

}

/\*\*

\* get the data at first node

\* **@return** first element or null if list is empty

\*/

**public** Object first() {

**if** (head == **null**) {

**return** **null**;

} **else** {

**return** head.data;

}

}

/\*\*

\* check if element existing in the list

\* **@param** element element to check

\* **@return** true if exists; otherwise, return false

\*/

**public** **boolean** contains(Object element) {

Node current = head;

//iterate the list to find

**while** (current != **null** && !current.data.equals(element)){

current = current.next;

}

//if current is not null, it contains element

**return** current != **null**;

}

/\*\*

\* get the number of elements in list

\* **@return** the number of elements in list

\*/

**public** **int** size(){

**int** numElements = 0;//number of elements

Node current = head;

//iterate the list

**while** (current != **null**){

numElements++;

current = current.next;

}

**return** numElements;

}

/\*\*

\* remove element

\* **@param** element element to remove

\* **@return** true if element in list; otherwise, return false

\*/

**public** **boolean** remove(Object element) {

**boolean** found = **false**; //element in list?

Node prev = **null**; //previous node of current

Node current = head;

//iterate the list

**while** (current != **null** && !current.data.equals(element)){

prev = current;

current = current.next;

}

**if** (current != **null**){//found

found = **true**;

//remove node

**if** (prev == **null**){//remove head node

head = head.next;

}**else**{

prev.next = current.next;//set the link to remove current

}

}

**return** found;

}

/\*\*

\* check if list is empty

\* **@return** true if list is empty; otherwise, return false

\*/

**public** **boolean** isEmpty(){

**return** head == **null**;

}

}

Stack.java

/\*\*

\* Stack class represents the stack by using

\* SinglyListList

\* It uses the head node of SinglyListList object

\* as the top node of stack

\*

\*/

**public** **class** Stack{

/\*\*

\* stack data

\*/

**private** SinglyLinkedList data = **new** SinglyLinkedList();

/\*\*

\* retrieve, not remove the top of stack

\* **@return** data at top or null if stack is empty

\*/

**public** Object peek(){

**return** data.first();

}

/\*\*

\* retrieve, and remove the top of stack

\* **@return** data at top or null if stack is empty

\*/

**public** Object pop(){

**return** data.removeFirst();

}

/\*\*

\* push an element to stack

\*

\* **@param** e

\*/

**public** **void** push(Object e){

//put element as the first node of linked list

data.addToHead(e);

}

/\*\*

\* get the number of elements in stack

\* **@return** the number of elements in stack

\*/

**public** **int** size(){

**return** data.size();

}

/\*\*

\* check if stack is empty

\* **@return** true if stack is empty; otherwise, return false

\*/

**public** **boolean** isEmpty(){

**return** data.isEmpty();

}

}

Utils.java

**import** java.util.Scanner;

/\*\*

\* utility class

\*/

**public** **class** Utils {

/\*\*

\* create new Scanner object

\*/

**private** **static** Scanner *scanner* = **new** Scanner(System.***in***);

/\*\*

\* read an integer between minimum and maximum

\* **@param** prompt

\* **@param** min minimum

\* **@param** max maximum

\* **@return** value in range minimum, maximum inclusive

\*/

**public** **static** **int** readInt(String prompt, **int** min, **int** max){

**int** number;//valid number

System.***out***.print(prompt);

//loop until user enter an integer

**while** (**true**){

**try**{

number = Integer.*parseInt*(*scanner*.nextLine());

**if** (number < min || number > max){

System.***out***.printf("Please enter number between %d and %d inclusive. Try again: ", min, max);

}**else**{

**break**;

}

}**catch**(NumberFormatException e){

System.***out***.print("Invalid input. Try again: ");

}//end try

}//end while

**return** number;

}

/\*\*

\* get the positions cells that knight can move to

\* **@param** chessboard chess board

\* **@param** Q a position

\* **@return** number of cells (not yet visited) and knight can move to

\*/

**public** **static** SinglyLinkedList getNeighbors(**int**[][] chessboard, Position Q){

SinglyLinkedList list = **new** SinglyLinkedList();

//row and column of Q

**int** i = Q.getRow();

**int** j = Q.getColumn();

**if** (*isValid*(chessboard, i - 2, j + 1) && chessboard[i - 2][j + 1] == -1){

list.add(**new** Position(i - 2, j + 1));

}

**if** (*isValid*(chessboard, i - 1, j + 2) && chessboard[i - 1][j + 2] == -1){

list.add(**new** Position(i - 1, j + 2));

}

**if** (*isValid*(chessboard, i + 1, j + 2) && chessboard[i + 1][j + 2] == -1){

list.add(**new** Position(i + 1, j + 2));

}

**if** (*isValid*(chessboard, i + 2, j + 1) && chessboard[i + 2][j + 1] == -1){

list.add(**new** Position(i + 2, j + 1));

}

**if** (*isValid*(chessboard, i + 2, j - 1) && chessboard[i + 2][j - 1] == -1){

list.add(**new** Position(i + 2, j - 1));

}

**if** (*isValid*(chessboard, i + 1, j - 2) && chessboard[i + 1][j - 2] == -1){

list.add(**new** Position(i + 1, j - 2));

}

**if** (*isValid*(chessboard, i - 1, j - 2) && chessboard[i - 1][j - 2] == -1){

list.add(**new** Position(i - 1, j - 2));

}

**if** (*isValid*(chessboard, i - 2, j - 1) && chessboard[i - 2][j - 1] == -1){

list.add(**new** Position(i - 2, j - 1));

}

**return** list;

}

/\*\*

\* check if row, column is valid indices

\* **@param** chessboard chess board

\* **@param** row

\* **@param** column

\* **@return** valid or not

\*/

**public** **static** **boolean** isValid(**int**[][] chessboard, **int** row, **int** column){

**return** row >= 0 && row < chessboard.length &&

column >= 0 && column < chessboard.length;

}

}

WarnsdorffAlgorithm.java

/\*\*

\* Warnsdorff's algorithm

\* that solves the number of steps

\* in Knight's tour problem

\*

\* Input: initial position, empty chess board

\* Output: chess board is loaded with number of steps: 1, 2,...

\*/

**public** **class** WarnsdorffAlgorithm {

/\*\*

\* initial position

\*/

**private** Position initialPosition;

/\*\*

\* chess board

\*/

**private** **int**[][] chessboard;

/\*\*

\* number of steps

\*

\* <= 64

\*/

**private** **int** numSteps;

/\*\*

\* last position in this algorithm

\*/

**private** Position lastPostion;

/\*\*

\* constructor

\*

\* **@param** initialPosition initial position

\* **@param** chessboard reference to chess board

\* **@param** numSteps number of steps

\*/

**public** WarnsdorffAlgorithm(Position initialPosition, **int**[][] chessboard, **int** numSteps){

**this**.initialPosition = initialPosition;

**this**.chessboard = chessboard;

**this**.numSteps = numSteps;

}

/\*\*

\* solve the problem

\*

\* algorithm:

\*

\* Set P is initial position on the board

\* Set location of P is 1 in the chess board

\* do

\* Choose P is the minimum position from set of positions accessible from P

\* until number of steps

\*/

**public** **void** solve(){

Position P = initialPosition;

**int** move = 1; //current move

chessboard[P.getRow()][P.getColumn()] = move;

move++;

lastPostion = P;

**for** (**int** i = 1; P != **null** && i < numSteps; i++){

P = getMin(P);

**if** (P != **null**){

chessboard[P.getRow()][P.getColumn()] = move;

move++;

lastPostion = P;

}

}

}

/\*\*

\* get the minimum position from set of positions accessible from P

\* **@param** P a position

\* **@return** position with minimum accessibility

\*/

**public** Position getMin(Position P){

Position min = **null**;

**int** minNeighboards = 0;

SinglyLinkedList neighbors = Utils.*getNeighbors*(chessboard, P);

**while** (!neighbors.isEmpty()){

Position pos = (Position)neighbors.removeFirst();

**int** numNeighbors = countNeighbors(pos);

**if** (min == **null** || minNeighboards > numNeighbors){

min = pos;

minNeighboards = numNeighbors;

}

}

**return** min;

}

/\*\*

\* get the number of cells that knight can move to

\* **@param** Q a position

\* **@return** number of cells (not yet visited) and knight can move to

\*/

**public** **int** countNeighbors(Position Q){

**return** Utils.*getNeighbors*(chessboard, Q).size();

}

/\*\*

\* get last position

\* **@return** the lastPostion

\*/

**public** Position getLastPostion() {

**return** lastPostion;

}

}

# Output

## Run 1

Please enter row index [1-8]. Enter 0 to exit: 1

Please enter column index [1-8]. Enter 0 to exit: 2

Please enter row index [1-8]. Enter 0 to exit: 3

Please enter column index [1-8]. Enter 0 to exit: 4

Please enter row index [1-8]. Enter 0 to exit: 5

Please enter column index [1-8]. Enter 0 to exit: 6

Please enter row index [1-8]. Enter 0 to exit: 7

Please enter column index [1-8]. Enter 0 to exit: 8

Please enter row index [1-8]. Enter 0 to exit: 0

Current initial positions:

1. (row = 1, column = 2)

2. (row = 3, column = 4)

3. (row = 5, column = 6)

4. (row = 7, column = 8)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 4

Initial position: row = 1, column = 2

4 1 6 21 28 61 16 19

7 22 3 60 17 20 29 62

2 5 24 27 50 59 18 15

23 8 35 58 25 56 63 30

36 45 26 51 34 49 14 55

9 42 39 46 57 54 31 64

40 37 44 11 52 33 48 13

43 10 41 38 47 12 53 32

Initial position: row = 3, column = 4

34 15 32 51 2 17 22 57

31 50 35 16 59 56 3 18

14 33 52 1 64 21 58 23

43 30 49 36 55 60 19 4

38 13 42 53 20 63 24 61

29 44 37 48 41 54 5 8

12 39 46 27 10 7 62 25

45 28 11 40 47 26 9 6

Initial position: row = 5, column = 6

6 23 8 45 4 21 18 55

9 44 5 22 35 54 3 20

24 7 36 51 46 19 56 17

37 10 43 60 49 34 53 2

62 25 50 47 52 1 16 57

11 38 61 42 59 48 33 30

26 63 40 13 28 31 58 15

39 12 27 64 41 14 29 32

Initial position: row = 7, column = 8

32 13 30 49 60 15 56 47

29 50 33 14 45 48 61 16

12 31 42 59 62 57 46 55

41 28 51 34 53 44 17 64

36 11 40 43 58 63 54 19

27 24 35 52 39 18 3 6

10 37 22 25 8 5 20 1

23 26 9 38 21 2 7 4

## Run 2

Please enter row index [1-8]. Enter 0 to exit: 1

Please enter column index [1-8]. Enter 0 to exit: 1

Please enter row index [1-8]. Enter 0 to exit: 0

Current initial positions:

1. (row = 1, column = 1)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 1

Please enter row index [1-8]. Enter 0 to exit: 1

Please enter column index [1-8]. Enter 0 to exit: 1

You entered that position already

Current initial positions:

1. (row = 1, column = 1)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 1

Please enter row index [1-8]. Enter 0 to exit: 1

Please enter column index [1-8]. Enter 0 to exit: 2

Current initial positions:

1. (row = 1, column = 1)

2. (row = 1, column = 2)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 1

Please enter row index [1-8]. Enter 0 to exit: 3

Please enter column index [1-8]. Enter 0 to exit: 4

Current initial positions:

1. (row = 1, column = 1)

2. (row = 1, column = 2)

3. (row = 3, column = 4)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 2

Current initial positions:

1. (row = 1, column = 1)

2. (row = 1, column = 2)

3. (row = 3, column = 4)

Please enter a position: 44

Please enter number between 1 and 3 inclusive. Try again: 1

Please enter row index [1-8]. Enter 0 to exit: 3

Please enter column index [1-8]. Enter 0 to exit: 4

You entered that position already

Current initial positions:

1. (row = 1, column = 1)

2. (row = 1, column = 2)

3. (row = 3, column = 4)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 2

Current initial positions:

1. (row = 1, column = 1)

2. (row = 1, column = 2)

3. (row = 3, column = 4)

Please enter a position: 5

Please enter number between 1 and 3 inclusive. Try again: 2

Please enter row index [1-8]. Enter 0 to exit: 5

Please enter column index [1-8]. Enter 0 to exit: 6

Current initial positions:

1. (row = 1, column = 1)

2. (row = 5, column = 6)

3. (row = 3, column = 4)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 3

Current initial positions:

1. (row = 1, column = 1)

2. (row = 5, column = 6)

3. (row = 3, column = 4)

Please enter a position2

Current initial positions:

1. (row = 1, column = 1)

2. (row = 3, column = 4)

1. add

2. edit

3. remove

4. exit

Please enter your selection? 4

Initial position: row = 1, column = 1

1 16 27 22 3 18 59 62

26 23 2 17 58 61 4 19

15 28 25 56 21 54 63 60

24 41 30 49 44 57 20 5

29 14 37 42 55 48 53 64

34 31 40 45 50 43 6 9

13 38 33 36 11 8 47 52

32 35 12 39 46 51 10 7

Initial position: row = 3, column = 4

34 15 32 51 2 17 22 57

31 50 35 16 59 56 3 18

14 33 52 1 64 21 58 23

43 30 49 36 55 60 19 4

38 13 42 53 20 63 24 61

29 44 37 48 41 54 5 8

12 39 46 27 10 7 62 25

45 28 11 40 47 26 9 6

## Run 3

Please enter row index [1-8]. Enter 0 to exit: 0

The initial positions list is empty

1. add

2. edit

3. remove

4. exit

Please enter your selection? 4

There is no initial position