רקורסיה

רקורסיה (recursion) היא טכניקה המשמשת הן להגדרת ישויות ומושגים (בעיקר במתמטיקה ובמדעי המחשב) והן לפתרון בעיות. הרעיון שמאחורי טכניקה זו היא שימוש במושג עצמו כדי להגדיר אותו, או שימוש בבעיה עצמה כדי לפתור אותה. המילה recursion נלקחה מהפועל recur שפירושו להתרחש בשנית re-occur. אנו יכולים למצוא רקורסיה בהקשרים רבים ושונים. ניתן לדבר על מבנים רקורסיבים, תיאורים רקורסיבים, הגדרות רקורסיביות, אלגוריתמים רקורסיבים ועוד. נראה כמה דוגמאות:

- מבנה רקורסיבי הוא מבנה המכיל בתוכו מבנה פשוט יותר מאותו סוג. לדוגמא, הבובות הרוסיות המצוירות, שכאשר פותחים כל אחת מהן מגלים בתוכה בובה זהה אך קטנה יותר. בסוף מגיעים לבובה שאי אפשר לפתוח.
- נוכל לתאר בצל בדרך שונה מהמקובל, דרך רקורסיבית. נתאר בצל כגלד שבתוכו בצל. עם כל הסרה של גלד מתגלה בצל נוסף, עד שמגיעים לגלד האחרון שאחריו אין כלום.

גם מחרוזת תווים נוכל לתאר באופן רקורסיבי: מחרוזת תווים סופית היא ריקה או שהיא תו ולאחריו מחרוזת תווים. שימו לב שאנו משתמשים במושג מחרוזת כדי לתאר מחרוזת. המחרוזת שנמצאת אחרי התו היא בהכרח קצרה יותר מהמחרוזת הראשונית. לדוגמא, המחרוזת "אבג" היא התו "א" ולאחריו המחרוזת "בג". המחרוזת "בג" היא התו "ב" ולאחריו המחרוזת "ג". המחרוזת "ג" היא התו "ג" ולאחריו המחרוזת הריקה.

- הגדרה רקורסיבית מפורסמת מאוד היא "**יהודי** הוא מי שאמו **יהודיה** או שגויר כהלכה". גם כאן אנו משתמשים במושג יהודי כדי להגדיר יהודי. בכל דור שאנו הולכים אחורה אנו מתקרבים אל אברהם אבינו שהיה הגֵר הראשון.

אפשר גם להתייחס להגדרה שהבאנו למשפט בשפה Java. משפט התנאי הוא חלק מהגדרת משפט (statement) ב- Java, ואף הוא מוגדר בעזרת שימוש במושג משפט.

selection statement

→ **if** → **(** → *boolean expression* → **)** →

→ *statement* → **;**

בכל המקרים הללו אנו רואים דפוס קבוע של שני מרכיבים, צעד הרקורסיה ותנאי עצירה. תנאי העצירה בדוגמאות שהבאנו היו הבובה האחרונה שלא נפתחת, הבצל הריק, מחרוזת התווים הריקה והגֵר (אברהם אבינו או מישהו אחר שהתגייר). צעדי הרקורסיה היו פתיחת הבובה ומציאת הבובה הקטנה יותר, קילוף הגלד וגילוי הבצל שבו, "הורדת" ראש המחרוזת וגילוי

מחרוזת תווים קצרה יותר ופנייה לדור הקודם אל האם היהודייה. שימו לב שצעד הרקורסיה מקרב אותנו תמיד לכוון תנאי העצירה.

נגדיר באופן רקורסיבי את המושג **צאצא**: צאצא של אדם הוא בנו (בתו) של אותו אדם או צאצא של בנו (בתו). חשבו, מהו צעד הרקורסיה ומהו תנאי העצירה בהגדרה זו?

כפי שאמרנו, רקורסיה היא גם טכניקה לפתרון בעיות. הרעיון בטכניקה זו הוא לחלק את הבעיה המקורית לתת-בעיות פשוטות יותר, אך בעלות מבנה זהה, עד שהן כל-כך פשוטות שהן ניתנות לפתרון בדרך ישירה.

לדוגמא, נניח שברצוננו לפרוט שטר של מאה שקלים למטבעות הקטנים או שווים לשקל. במקום לחפש מישהו שיש לו מאה מטבעות של שקל (או יותר אם מדובר במטבעות קטנים יותר), יקל עלינו למצוא אדם שיש לו שני שטרות בני חמישים שקלים. אחר-כך ננסה לפרוט כל אחד מהשטרות האלו לשטרות קטנים יותר או למטבעות של עשרה שקלים. וכך הלאה עד שנצליח לפרוט את כל הסכום שרצינו. בפתרון בעיה זו השתמשנו באותה בעיה (פריטת סכום כסף) אלא שהסכום היה קטן יותר. צעד הרקורסיה הוא בדיוק זה הפותר את הבעיה בסכום קטן יותר. תנאי העצירה מתקיים כאשר המטבעות קטנים או שווים לשקל.

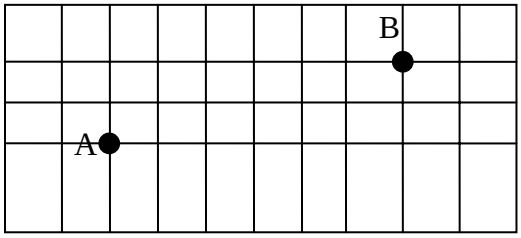
דוגמא נוספת היא בניסיון לפרק מספר לגורמיו הראשונים. נחלק את המספר n לשני גורמים, לאו דווקא ראשוניים, a ו-b כך ש- $n = a*b$. אחר-כך נבצע תהליך זה גם לגבי הגורמים a ו-b אם אינם ראשוניים. וכך נמשיך עד שכל הגורמים יהיו ראשוניים. גם כאן, צעד הרקורסיה הוא החלוקה לשני גורמים של מספר קטן יותר. גבול הרקורסיה - תנאי העצירה - הוא כשמגיעים למספרים ראשוניים.

אנו יכולים לסכם ולומר כי **אלגוריתם** לפתרון בעיה הוא **רקורסיבי**, אם הוא מאפשר לפתור את הבעיה בדרך המצריכה את פתרון אותה בעיה עצמה - במקרה (או מקרים) פשוט (פשוטים) יותר, וש"פשטות" המקרים "נעצרת" במקום מסוים בו ניתן לפתור את הבעיה ישירות. תכנית מחשב או תת-תכנית (פונקציה/שיטה) מכונה רקורסיבית אם היא מיישמת אלגוריתם רקורסיבי.

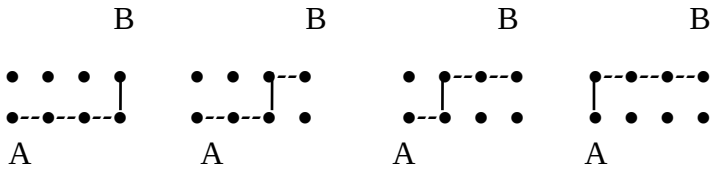
בהרצאה ניתנו דוגמאות רבות לשיטות רקורסיביות. כאן נוסיף כמה דוגמאות מורכבות.

דוגמא 1

נניח שהסריג הבא מסמן רחובות בצורת שתי וערב:



ב*מסלול צפון-מזרח* מנקודה אחת על הסריג לנקודה אחרת בו אפשר ללכת רק לכוון צפון (למעלה) ולכוון מזרח (ימינה). לדוגמא, ישנם ארבעה מסלולי צפון-מזרח מהנקודה A לנקודה B בסריג, כפי שרואים באיור הבא:



נכתוב פונקציה רקורסיבית הסופרת את מסלולי צפון-מזרח יש מנקודה A לנקודה B בסריג. כדי לפשט את הדברים, נניח כי הנקודה A היא תמיד (0, 0), והנקודה B היא (x, y). כלומר, בהינתן שני מספרים x ו- y, עלינו לחשב את מספר המסלולים מ- (0,0) ל- (x, y).

מהי אסטרטגית הפתרון? ובכן, כדי להגיע אל הנקודה (x, y) במסלול צפון-מזרח בהכרח עלינו לעבור בסוף המסלול באחת משתי הנקודות הבאות: (x, y-1) כאשר באים מלמטה, או (x-1, y) כאשר באים משמאל. כלומר מספר כל המסלולים שמגיעים אל (x, y) שווה למספר המסלולים שמגיעים ל- (x, y-1) ועוד מספר המסלולים המגיעים ל- (x-1, y).

הנה השיטה המממשת אלגוריתם זה:

```
int count (int x, int y)
// count the number of paths from (0,0) to (x,y).
{
   if ((x==0) || (y==0))
       return 1;
   return (count (x-1, y) + count (x, y-1));
}
```

דוגמא 2 - יציאה ממבוך

נניח שלפנינו המבוך הבא, כאשר הסימן '-' מסמן מקום פתוח והסימן '#' מסמן קיר שאי אפשר לעבור. נניח שעכבר נמצא במרכז המבוך.

```
# # # - #
# - # - #
# - - - #
# - # - -
# # # # #
```

ברצוננו לכתוב תכנית שתוציא את העכבר מהמבוך (כלומר, תצייר את כל המסלולים בהם הוא יכול לצאת מהמסגרת.) אנו מניחים שאפשר ללכת ימינה ושמאלה, למעלה ולמטה, אך לא באלכסון. הגעה למקום פתוח במסגרת נחשבת ליציאה מהמבוך.

אסטרטגית הפתרון תהיה לנסות לצאת מנקודת המוצא למקום פתוח באחד מארבעת הכיוונים, אם יש כזה. משם שוב להמשיך ולנסות לעבור למקום פתוח באחד מארבעת הכוונים, וכך הלאה עד למסגרת. אם נתקעים במקום ממנו אין המשך, חוזרים צעד אחד אחורה ומנסים שוב לצאת ממנו לכוון אחר. אסטרטגיה זו נקראת **עקיבה לאחור** (backtracking).

אנו נאחסן את המבוך במערך דו-ממדי. נקרא מהקלט את צורת המבוך תו אחרי תו לתוך מערך של תווים. נקרא מהקלט גם את נקודת ההתחלה בה נמצא העכבר, כשתי הקואורדינטות של המערך. אנו מניחים כי העכבר מתחיל את דרכו במיקום פתוח. הדרך החוצה תוצג על הפלט בעזרת התו 's'.

השיטה exit מקבלת כפרמטר נקודה במבוך (על-ידי שתי הקואורדינטות שלה), ומחזירה 1 אם זו נקודה על המסגרת ו- 0 אחרת.

השיטה findWayOut היא השיטה הרקורסיבית, והיא מבצעת את החיפוש אחר היציאה מהמבוך. היא מקבלת כפרמטרים את המערך, ואת הנקודה העכשווית בה נמצא העכבר. בתחילה, השיטה מעתיקה את המערך למערך עזר, ומסמנת במערך העזר את המקום בו נמצא העכבר עם הסימן 's'. אם המקום נמצא במסגרת, סימן שהגענו ליציאה, והשיטה מדפיסה את המערך כפתרון. אם לא, היא בודקת את ארבעת השכנים מימין ומשמאל, מלמעלה ומלמטה, אחד אחרי השני, **ובכל אחד מהם**, אם המקום פתוח היא קוראת לעצמה ברקורסיה. הקריאה נעשית לכל ארבעת הצדדים; גם אם צד אחד פתוח ממשיכים לבדוק את האחרים. זה מה שמאפשר את העקיבה לאחור, וכן גורם לרקורסיה לסמן את כל המסלולים האפשריים.

הנה התכנית המבצעת זאת. שימו לב שכאן לא כתבנו את הכנסת התווים למערך, ואת קליטת המיקום הראשוני של העכבר.

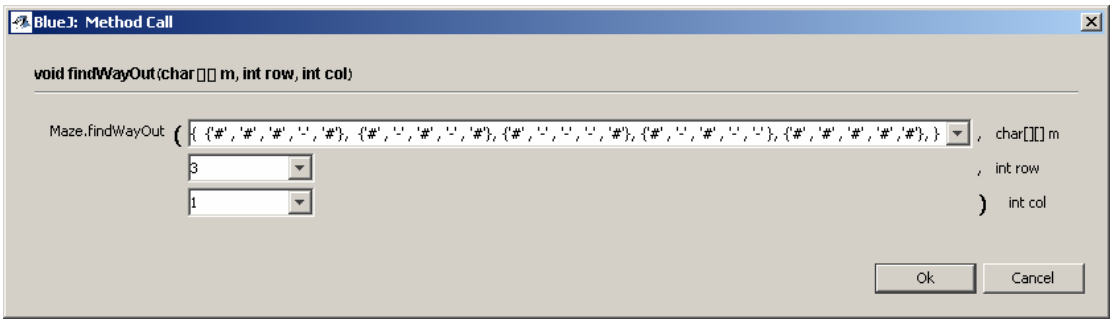
```
/* This program finds out the possible paths to go out of a
 * maze.
 * The input is the mouse position (remember that columns and
 * rows start at 0) and the maze pattern. The program uses
 * backtracking strategy.
*/
public class Maze
{
   private final static int MAX_ROW = 5;
   private final static  int MAX_COLUMN = 5;
   private final static  char OPEN = '-';
   private final static  char WAY_OUT = 's';
```

```
/*
private static char maze [][] =
      {  {'#' , '#' , '#' , '-' , '#'},
         {'#' , '-' , '#' , '-' , '#'},
         {'#' , '-' , '-' , '-' , '#'},
         {'#' , '-' , '#' , '-' , '-'},
         {'#' , '#' , '#' , '#' , '#'}, } ;
*/

/*
 * Returns true if position is on the outline,
 * else – returns false.
*/
private static boolean exit (int row, int col)
{
   return (row==0) || (row==MAX_ROW-1) ||
          (col==0) || (col==MAX_COLUMN-1);
}

/*
 * Prints the maze (matrix) as it is.
*/
public static void print (char [][] maze)
{
   for (int i=0; i<MAX_ROW; i++)
   {
      for (int j=0; j<MAX_COLUMN; j++)
          System.out.print(maze[i][j] + "\t");
      System.out.println();
   }
   System.out.println();
}
```

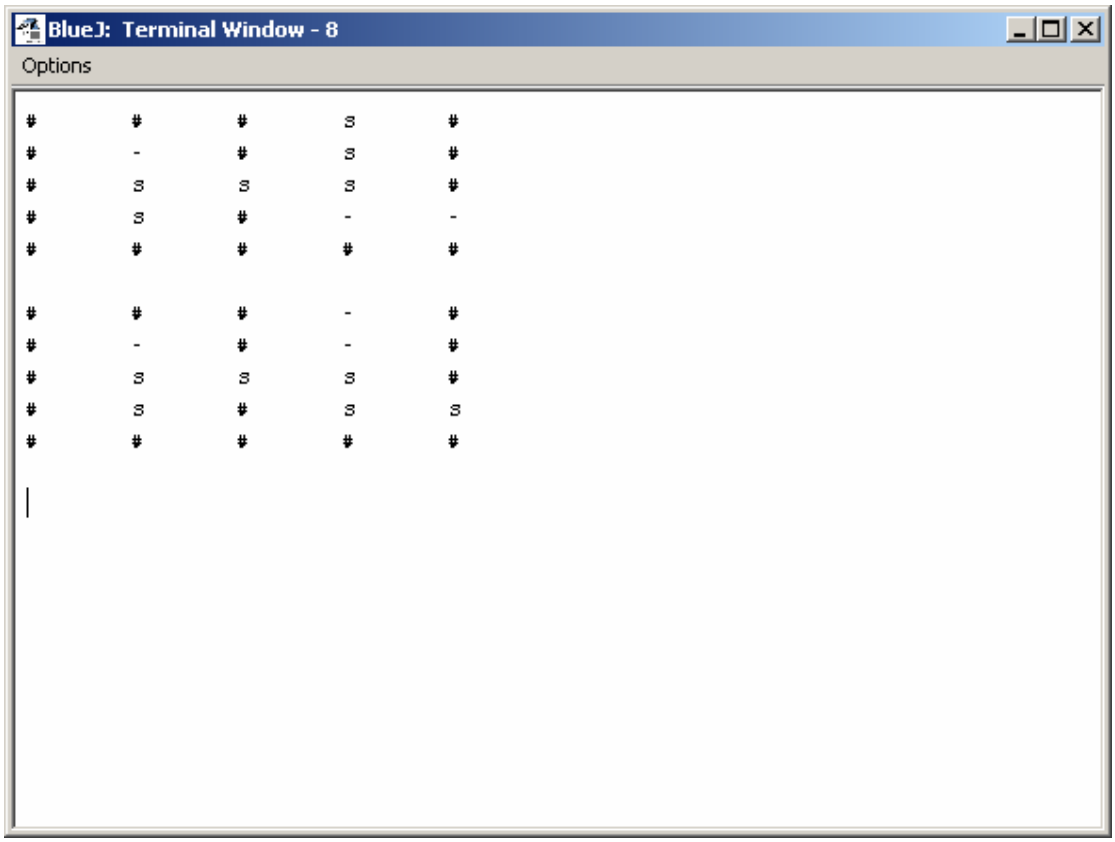
```
/* This is a recursive method that looks for a path out of the
 * maze. The parameters are the maze and the current position
 * of the mouse.
*/
public static void findWayOut (char [][] m, int row, int col)
{
   // the temporary (local) array
   char [][] maze = new char [MAX_ROW][MAX_COLUMN];

   // copy the array given as a parameter to a local array.
   for (int i=0; i<MAX_ROW; i++)
      for (int j=0; j<MAX_COLUMN; j++)
          maze [i][j] = m[i][j];

   // mark the current position as a point on the path.
   maze[row][col] = WAY_OUT;

   // if the position of the mouse is on the outline of the
   // maze – print the temporary matrix that contains the path
   // out of the maze.
   if (exit (row, col))
   {
      print (maze);
   }
   else
   // the mouse is not at the exit of the maze seek for its
   // way out with four directions:
   {
     if (maze[row-1][col] == OPEN)
         findWayOut (maze, row-1, col);
     if (maze[row][col+1] == OPEN)
         findWayOut (maze, row, col+1);
     if (maze[row+1][col] == OPEN)
         findWayOut (maze, row+1, col);
     if (maze[row][col-1] == OPEN)
        findWayOut (maze, row, col-1);
   }
}
```

הנה דוגמת להרצת התכנית על הקלט שלעיל, כאשר העכבר נמצא בשורה 3 ובעמודה 1.

זו הקריאה לשיטה findWayOut :



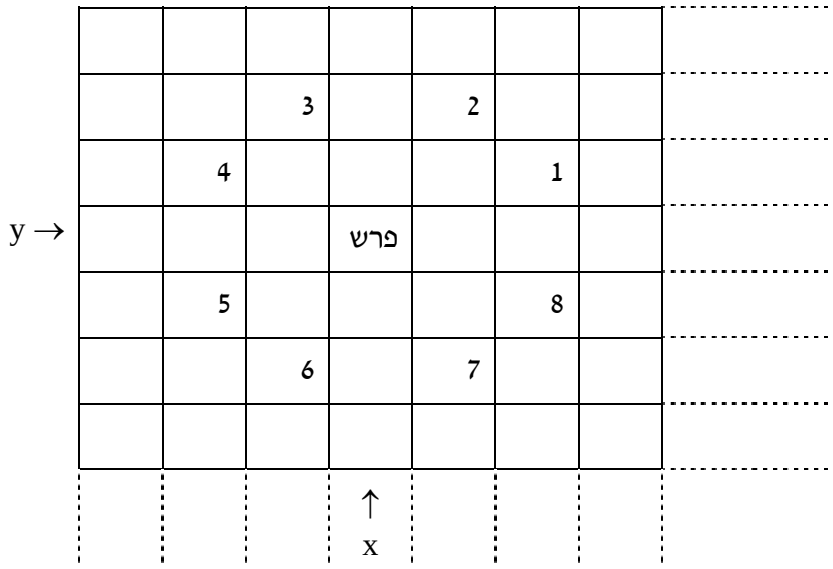
וזהו מסך הפלט שקיבלנו :



דוגמא 3 - מסעות הפָּרָשׁ בלוח

נתון לוח ריבועי בן n שורות ו- n עמודות (סה״כ n^2 משבצות).

פָּרָשׁ (knight) העומד על משבצת מסוימת יכול להגיע לכל אחת מהמשבצות המסומנות במספרים מ- 1 עד 8 להלן. הפרש עצמו ממוקם בשורה y ובעמודה x (כאשר מתקיים $1 \le y \le n,\ 1 \le x \le n$).



מסע פרש מורכב ממעבר על שתי משבצות רצופות בכוון מאוזן ואחת בכוון מאונך, או שתי משבצות רצופות בכוון מאונך ואחת בכוון מאוזן. שימו לב כי אם הפרש נמצא על אחת המשבצות שבמסגרת הלוח או קרוב לה, הוא אינו יכול לבצע את כל שמונת המסעות.

אנו נכתוב תכנית המקבלת נקודת מוצא של הפרש (על-ידי קליטת שני מספרים שלמים המציינים שורה ועמודה כמצב התחלתי). התכנית מחפשת מסלול רציף של תנועות פרש, אשר יעבור דרך **כל** משבצות הלוח, ופעם אחת **בדיוק** בכל משבצת.

אם אין פתרון, נדפיס הודעה מתאימה על כשלון.

אם נמצא פתרון, נדפיס את הלוח כאשר תוכן משבצת המוצא יהיה 1, תוכן המשבצת הבאה אליה התקדם הפרש יהיה 2, וכן הלאה עד n^2. יתכנו מצבים בהם יש יותר מפתרון אחד. אנו נדפיס את כל הפתרונות האפשריים.

אסטרטגית הפתרון תהיה כזאת: נצא מנקודת ההתחלה (x, y), וננסה למצוא משבצת אחת אליה הפרש יכול להגיע. ממשבצת זו ננסה למצוא מסלול שיכסה את יתרת הלוח. אם אי אפשר, נבחר באופן חלופי במשבצת אחרת אליה יכול להגיע הפרש מ- (x, y). יש לכל היותר 7 משבצות כאלה פרט למשבצת שכבר נתגלתה כלא מתאימה.

מכאן ברור שהפתרון לבעיה יהיה באופן רקורסיבי. צריך לזכור גם שאם נסוגים ממסלול שנמצא לא מוצלח, יש להביא את הלוח למצב שהיה לפני ניסיון הנפל. הרקורסיה עוזרת לנו לעשות זאת, כיון שהיא שומרת במחסנית הזיכרון את הערכים שהיו לפני הקריאה הרקורסיבית. גם הפעם משתמשים בשיטת העקיבה לאחור (backtracking).

הנה תכנית הפותרת את הבעיה. שימו לב שלא הוספנו הודעה במקרה שאין פתרון (למשל כאשר MAX_ROW = MAX_COLUMN = 4). הוסיפו זאת בעצמכם. לא כדאי להריץ את התכנית למערך בגודל מעל 5×5:

```
import java.util.Scanner;

public class Knight
{

   public final static int MAX_ROW = 5;
   public final static  int MAX_COLUMN = 5;

   public static int [][] table =
        new int [MAX_ROW][MAX_COLUMN];

   public static void main (String [] args)
   {

        int x, y;

        Scanner scan = new Scanner(System.in);

        System.out.println ( "Enter the starting point: ");
        x = scan.nextInt();
        y = scan.nextInt();

        table[x][y] = 1;

        printTable (table);
        System.out.println ( "The solution is: ");
        nextMove (table,1, x, y);
   }


   private static boolean outOfRange (int row, int col)
   // Returns true iff the position is out of the board's
   // range.

   {
      return (row<0) || (row>=MAX_ROW) ||
             (col<0) || (col>=MAX_COLUMN);
   }
```

```
public static void nextMove (int [][] table ,int num,
                             int row, int col)
/* The recursive method that finds the next available move.
 * Parameters: the table, the number of the current move and
 * the current position.
*/
{

    int [][] t = new int [MAX_ROW][MAX_COLUMN];

   // Copy the current table to a temporary one.
   for (int i=0; i<MAX_ROW; i++)
      for (int j=0; j<MAX_COLUMN; j++)
          t [i][j] = table[i][j];

   // assign the current position with the current move.
   t[row][col] = num;

   // if the current move is the last one, print the board.

   if (num == MAX_ROW * MAX_COLUMN)
      printTable (t);

   else
   // looks for the next move in all possible directions.
   {
     if (!(outOfRange (row-1, col+2)) &&
          (t[row-1][col+2] == 0))
                nextMove (t, num+1, row-1, col+2);
     if (!(outOfRange (row-2, col+1)) &&
          (t[row-2][col+1] == 0))
                nextMove (t, num+1, row-2, col+1);
     if (!(outOfRange (row-1, col-2)) &&
          (t[row-1][col-2] == 0))
                nextMove (t, num+1, row-1, col-2);
     if (!(outOfRange (row-2, col-1)) &&
          (t[row-2][col-1] == 0))
                nextMove (t, num+1, row-2, col-1);
     if (!(outOfRange (row+1, col-2)) &&
          (t[row+1][col-2] == 0))
                nextMove (t, num+1, row+1, col-2);
     if (!(outOfRange (row+2, col-1)) &&
          (t[row+2][col-1] == 0))
                nextMove (t, num+1, row+2, col-1);
     if (!(outOfRange (row+1, col+2)) &&
            (t[row+1][col+2] == 0))
            nextMove (t, num+1, row+1, col+2);
     if (!(outOfRange (row+2, col+1)) &&
        (t[row+2][col+1] == 0))
            nextMove (t, num+1, row+2, col+1);
   } //of else

} // of the method nextMove

   public static void printTable (int  [][] table)
```

```
   // Prints the board.
   {
      for (int i=0; i<MAX_ROW; i++)
      {
        for (int j=0; j<MAX_COLUMN; j++)
           System.out.print(table[i][j] + "\t");
        System.out.println();
      }
      System.out.println();
   }

}  // of class knight
```

כאשר n=5 ונקודת ההתחלה היא (0,0), יש הרבה מאוד פתרונות אפשריים. הנה כמה דוגמאות:

1	16	11	6	21
10	5	20	15	12
17	2	13	22	7
4	9	24	19	14
25	18	3	8	23

1	16	21	8	25
22	17	24	15	20
17	2	11	6	9
12	23	4	19	14
3	18	13	10	5

1	18	13	22	7
12	23	8	19	14
17	2	21	6	9
24	11	4	15	20
3	16	25	10	5