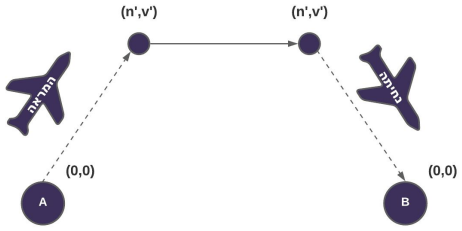
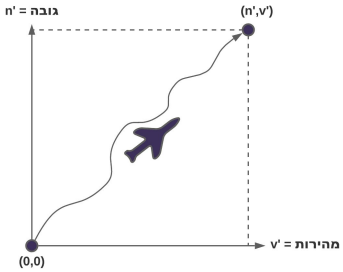
בעיית המטוס

תיאור הבעיה

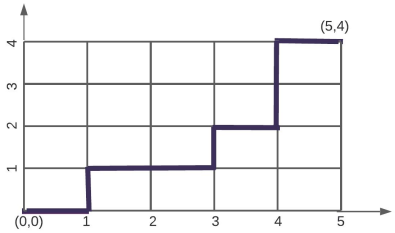
איך להמריא ולנחות בצורה הכי מהירה וזולה.
הטייס רוצה בשלב ההמראה לחסוך כמה שיותר דלק, אנחנו צריכים להגיד לטייס באיזה קצב לטוס בצורה שהכי תחסוך את הדלק.



הפרמטרים שלנו במקרה זה הם גובה ומהירות. אנחנו צריכים למצוא פונקציה שמשתנה בהתאם לשינויים של הפרמטרים בצורה הכי חסכונית שיש.אפשרות לפתרון הוא לפרק את הבעיה לתתי-תחומים לפי השינויים. אנחנו מתרגמים את הבעיה לצורה יותר "טכנית" וכך נוכל לפתור את זה בצורה פשוטה יותר.

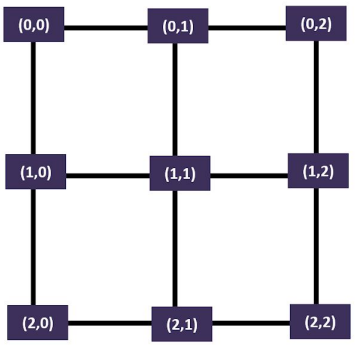


נתונה מטריצה המייצגת לוח משבצות $(n\ x\ m)$ כך שעבור כל מעבר בין 2 קודקודים סמוכים יש משקל כלשהו.
יש למצוא את המסלול עם העלות הנמוכה ביותר החל מקודקוד (0,0) עד קודקוד (n, m) כאשר מותר לצעוד למעלה או ימינה בלבד.המטריצה מיוצגת על-ידי מערך דו-מימדי של Nodes כאשר בכל Node יש x,y.



המטרה

למצוא את המסלול הקצר ביותר (המסלול בעל העלות המינימלית) מנקודה (0,0) לנקודה (M,N).כדי להיכנס לראש של מתכנת נהפוך את המטריצה שלנו בצורה הבאה כדי לפשט את הבעיה עוד יותר (האיור משמאל):



פתרון הבעיה

עבור כל אפשרות נשתמש באובייקט הבא *Node.java*:

- goRight, goDown - המחיר לגשת למטה או ימינה מהקודקוד הזה.
- entry - המחיר הטוב ביותר מנקודה (0,0) עד לקודקוד זה.
- numOfPaths - מספר המסלולים הקצרים ביותר עד לקודקוד הזה.
- entryFromTheEnd - כמו entry, רק שזה מהסוף עד לקודקוד.

```
public class Node {
   int goRight, goDown, entry, numOfPaths, entryFromTheEnd;
   public Node(int x, int y) {
       this.goRight = x;
       this.goDown = y;
       entry = 0;
       entryFromTheEnd = 0;
       numOfPaths = 1;
   }
}
```

❖ אפשרות א' - אלגוריתם חמדני.

- נבחר בכל שלב את המעבר עם העלות הנמוכה יותר.

נתבונן במטריצה $matrix$ $Node[][]$ באיור מלמטה:

נתחיל מנקודת ההתחלה שלנו (0,0) ונרצה למצוא את המסלול הזול ביותר ל-(1,1).

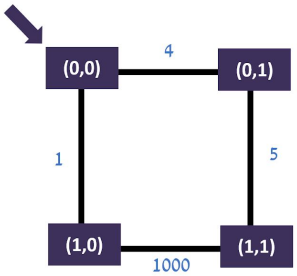
באופן חמדני נבחר "down" כי 1<4.

כעת אנו עומדים בנקודה (1,0), נותר לנו רק לבחור "right" ,

כלומר במשקל 1000 כדי להגיע ליעד.

קל לראות כמה החמדני לא יעיל לנו כי הגענו ליעד עם מסלול באורך 1001 ,

כלומר "$marix[1][1].entry \ == \ 1001$" כאשר היינו יכולים למצוא מסלול זול יותר באורך 9 ולכן החמדני בכלל לא רלוונטי עבור פתרון הבעיה.

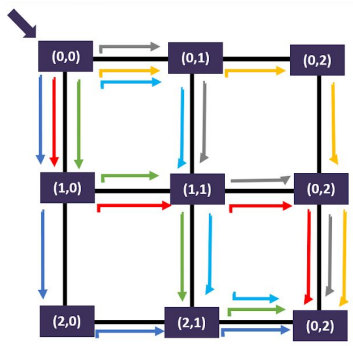


- **סיבוכיות**: $O(n+m)$ - מספר המעברים במסלול החל מקודקוד $(0,0)$ עד קודקוד (n,m) כאשר מותר לצעוד למטה או ימינה בלבד.
- **נכונות השיטה**: השיטה לא מחזירה את התשובה הנכונה כי ייתכן ואחרי מעבר זול יגיע מעבר יקר ולהיפך ולכן לפעמים יהיה כדאי לוותר על מעבר זול כדי להרוויח בהמשך.

❖ אפשרות ב' - חיפוש שלם.

- נייצר את כל המסלולים מהנקודה 1,1 ל m,n ונסכום את כל המשקלים של כל מסלול.
- ניקח את המסלול בעל העלות הנמוכה ביותר.
- נפתור ברקורסיה כאשר נתחיל מקודקוד (0,0) ונלך פעם אחת ימינה ופעם אחת למטה. נכנס לרקורסיה שוב ושוב עד שנגיע לקודקוד (N,M) וכך נעבור על כל המסלולים האפשריים.

כפי שאפשר לראות באיור מלמטה אמנם הגענו בהכרח לפתרון אבל עבור המטריצה הקטנה הזו ביצענו כמות גדולה של פעולות. עבור מטריצות גדולות יותר נגיע כבר למצב של פיצוץ קומבינטורי.



- **סיבוכיות**: $O(\binom{n+m}{n} \cdot (n+m))$ כלומר מספר המסלולים כפול מעבר על כל מסלול וחישוב העלות של המסלול.

❖ אפשרות ג' - תכנות דינאמי.

- בחיפוש השלם, עשינו בדיקות מיותרות, כי אם 2 מסלולים מגיעים לאותה נקודה ואחד מהם בעלות נמוכה יותר, ברור שניקח אותו ואין טעם להמשיך עם המסלול הארוך יותר.
- נייצר מטריצה שבה בכל תא $[i][j]$ נשמור את אורך המסלול הקצר ביותר מהנקודה (0,0) עד הנקודה (i,j) ונמלא את המטריצה באופן הבא:

העלות להגיע לנקודה (i,j) שווה למינימום:

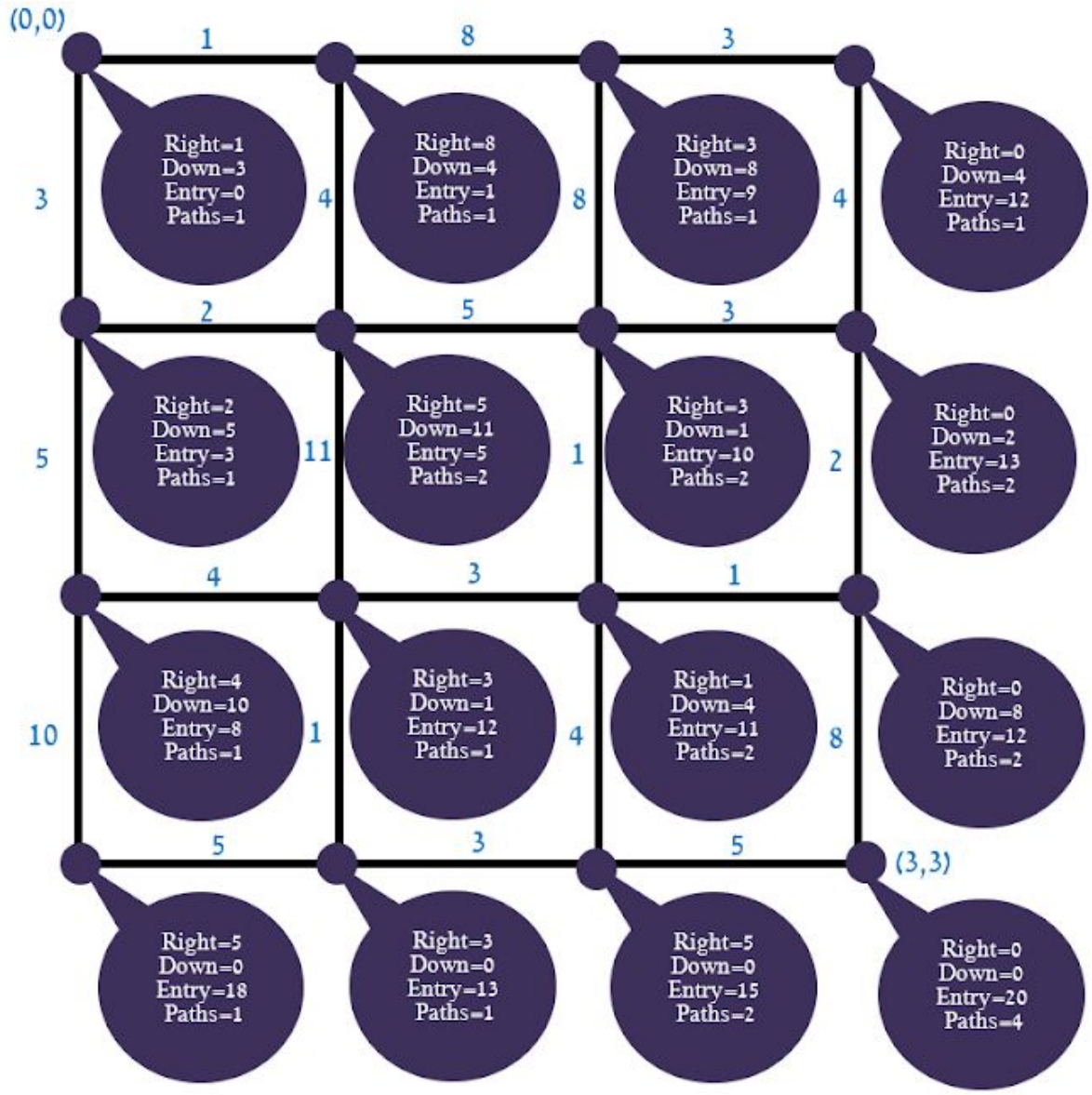
מבין העלות להגיע ל $(i,j-1)$ ואז ללכת ימינה, לבין העלות להגיע ל $(i-1,j)$ ואז ללכת למטה, כלומר :

```
int downPath = matrix[i - 1][j].entry + matrix[i - 1][j].goDown
int rightPath = matrix[i][j - 1].entry + matrix[i][j - 1].goRight;
matrix[i][j].entry = Math.min(rightPath, downPath);
```

נמלא תחילה עמודה ראשונה ושורה ראשונה (בדומה לבסיס אינדוקציה),

מן הסתם מספר המסלולים עבור כל קודקוד בפעולה זאת הוא מסלול 1 בלבד מכיוון שאני מוגבל להמשיך רק ימינה או למטה מכל קודקוד.

נתבונן באיור לדוגמא של matrix בגודל 3x3 כאשר המספרים על הצלעות הם המשקל ועבור כל ערך Entry נקבל את המחיר הזול ביותר מנקודה (0,0) עד לנקודה נוכחית שלה, בנוסף paths זה מספר המסלולים הזולים ביותר מנקודה (0,0) ועד לקודקוד זה:



יצירת המטריצה (תואם לציור מלמעלה) וחישוב ערכי Entry עבור כל צומת במטריצה.

לטובת נוחות המימוש, נדמה את הנתונים לשימושים שמוכרים לנו כבר, למשל עבור ההשמה הבאה:

`mat[0][0] = new Node(3,1);`

הערך 3 יהיה הערך לרדת למטה goDown במטריצה (כמו הערך i בלולאה כפולה על מערך דו מימדי).

הערך 1 יהיה הערך לימין goRight במטריצה (כמו הערך j בלולאה כפולה על מערך דו מימדי).

חשוב לציין כי עבור מקרים דומים ל- `mat[1][3] = new Node(2,0)` ימינה הוא 0 מכיוון שהגענו לקצה מימין ואין עוד לאן להמשיך משם.

```
public static void generateMatrix(Node[][] matrix) {

   for(int i = 1 ; i < matrix.length; i++) { // O(N)
       matrix[i][0].entry = matrix[i-1][0].entry + matrix[i-1][0].goDown;
   }
   for(int i = 1 ; i < matrix[0].length; i++) { // O(M)
       matrix[0][i].entry = matrix[0][i-1].entry + matrix[0][i-1].goRight;
   }
   for(int i = 1; i < matrix.length; i++) { // O(N*M)
       for(int j = 1; j < matrix[0].length; j++) {
           int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
           int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
           matrix[i][j].entry = Math.min(fromAbove,fromLeft);
       }
   }
}
public static void main(String[] args) {
   Node[][] mat = new Node[4][4];
   mat[0][0] = new Node(3,1);
   mat[0][1] = new Node(4,8);
   mat[0][2] = new Node(8,3);
   mat[0][3] = new Node(4,0);
   mat[1][0] = new Node(5,2);
   mat[1][1] = new Node(11,5);
   mat[1][2] = new Node(1,3);
   mat[1][3] = new Node(2,0);
   mat[2][0] = new Node(10,4);
   mat[2][1] = new Node(1,3);
   mat[2][2] = new Node(4,1);
   mat[2][3] = new Node(8,0);
   mat[3][0] = new Node(0,5);
   mat[3][1] = new Node(0,3);
   mat[3][2] = new Node(0,5);
   mat[3][3] = new Node(0,0);
   generateMatrix(mat);
}
```

↜ **סיבוכיות השיטה**: $O(n \cdot m)$ ממלאים את המטריצה לפי החוקיות.

יתרון נוסף של הפונקציה **generateMatrix** הוא האפשרות לחשב את מספר המסלולים הזולים ביותר עבור כל צומת (במקביל לבניית המטריצה) בתוך הלולאה של הפונקציה הזאת.

```
public static void generateMatrix(Node[][] matrix) {

   for(int i = 1 ; i < matrix.length; i++) { // O(N)
       matrix[i][0].entry = matrix[i-1][0].entry + matrix[i-1][0].goDown;
   }
   for(int i = 1 ; i < matrix[0].length; i++) { // O(M)
       matrix[0][i].entry = matrix[0][i-1].entry + matrix[0][i-1].goRight;
   }

   for(int i = 1; i < matrix.length; i++) { // O(N*M)
       for(int j = 1; j < matrix[0].length; j++) {
           int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
           int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
           matrix[i][j].entry = Math.min(fromAbove,fromLeft);

// addition:
           int leftPaths = matrix[i][j-1].numOfPaths;
           int abovePaths = matrix[i-1][j].numOfPaths;
           if(fromAbove > fromLeft) {
               matrix[i][j].numOfPaths = leftPaths;
           }
           else if(fromAbove < fromLeft) {
               matrix[i][j].numOfPaths = abovePaths;
           }
           else { // if they are equals.
               matrix[i][j].numOfPaths = abovePaths + leftPaths;
           }
       }
   }
}
```

מימוש **רקורסיבי** לבניית המטריצה

```
public static void generateRec(Node[][] matrix, int i, int j) {
   if(i == matrix.length)
       return;
   if(j == matrix.length) {
       generateRec(matrix,i+1,1);
   }
   else {
       int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
       int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
       matrix[i][j].entry = Math.min(fromAbove,fromLeft);

       int abovePaths = matrix[i-1][j].numOfPaths;
       int leftPaths = matrix[i][j-1].numOfPaths;
       if(fromAbove > fromLeft){
           matrix[i][j].numOfPaths = leftPaths;
       } else if(fromAbove < fromLeft) {
           matrix[i][j].numOfPaths = abovePaths;
       } else {
           matrix[i][j].numOfPaths = leftPaths + abovePaths;
       }
       generateRec(matrix,i,j+1);
   }
}

public static void generateMatrix(Node[][] matrix) {
   for(int i = 1 ; i < matrix.length; i++)
       matrix[i][0].entry = matrix[i-1][0].entry + matrix[i-1][0].goDown;
   for(int j = 1 ; j < matrix[0].length; j++)
       matrix[0][j].entry = matrix[0][j-1].entry + matrix[0][j-1].goRight;

   generateRec(matrix,1,1);
}
```

נמשיך לעבוד עם שיטת התכנות הדינאמי, ונציג עבורה פונקציות שימושיות עבור מסלולים:

- **getCornerPath** - מחזיר את המסלול הזול ביותר בין נקודת ההתחלה (0,0) לבין הנקודה (n,m).
- **getPath** - מחזיר את המסלול הזול ביותר עבור צומת i,j שנשלח לפונקציה.
- **getAllPaths** - מחזיר את המסלול הזול ביותר לכל הצמתים.

מימוש **אינדוקטיבי**:

```
public static String getCornerPath(Node[][] matrix) {
   return getPath(matrix,matrix.length-1,matrix[0].length-1);
}

public static String getPath(Node[][] matrix, int i, int j) {

   String direction = "("+i+","+(j)+")-> ";
   String ans = direction;
   while(i >= 1 && j >= 1) {
       int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
       int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
       if(fromAbove > fromLeft) {
           direction = "("+i+","+(j-1)+")-> ";
           ans = direction + ans;
           j--;
       } else {
           direction = "("+(i-1)+","+(j)+")-> ";
           ans = direction + ans;
           i--;
       }
   }
   direction = "(0,0)-> ";
   ans = direction + ans;

   return ans;
}

public static void getAllPaths(Node[][] matrix) {
   System.out.println("\n===== ALL PATHS ====");
   for(int i = 1; i < matrix.length; i++) {
       for(int j = 1 ; j < matrix[0].length; j++) {
           matrix[i][j].myShortestPath = getPath(matrix,i,j);
           System.out.println("Path of ("+i+","+j+") is [ "+
matrix[i][j].myShortestPath +" ]");
       }
   }
}
```

במימוש **רקורסיבי** זה קיימת גם פונקציה **getAllPaths** שמחזירה לא רק את המסלול הזול ביותר (יחיד) לכל הקודקודים בגרף אלא גם מחזירה את **כל** המסלולים הזולים ביותר עבור **כל** צומת (במידה ויש יותר ממסלול אחד כזה):

```
public static String getCornerPath(Node[][] matrix) {
   return getOnePath(matrix,matrix.length-1,matrix[0].length-1);
}

public static String getOnePath(Node[][] matrix, int i, int j) {
   String direction = "("+i+","+j+")-> ";
   return getOnePathRec(matrix,i,j) + direction;
}

private static String getOnePathRec(Node[][] matrix, int i, int j) {
   if(i == 0 || j == 0) {
       return "(0,0)-> ";
   }
   int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
   int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;

   if(fromAbove > fromLeft) {
       String direction = "("+i+","+(j-1)+")-> ";
       return getOnePathRec(matrix,i,j-1) + direction;
   }

   String direction = "("+(i-1)+","+j+")-> ";
   return getOnePathRec(matrix,i-1,j) + direction;
}

public static void getAllPaths(Node[][] matrix) {
   System.out.println("\n===== ALL PATHS ====");
   getAllRec(matrix,1,1);
}

private static void getAllRec(Node[][] matrix, int i, int j) {
   if(i == matrix.length)
       return;
   if(j == matrix[0].length) {
       getAllRec(matrix,i+1,1);
   }else {
       Vector<String> paths = new Vector<>();
       getPaths(matrix,i,j,"("+i+","+j+")",paths);
       String ans = "";
       for(String str : paths) {
           ans = ans + "\n" +  str;
       }
       matrix[i][j].myShortestPath = ans;
```

```
        System.out.println("Path of ("+i+","+j+") is [ "+
matrix[i][j].myShortestPath +"\n]");
        getAllRec(matrix,i,j+1);
    }
}

private static void getPaths(Node[][] matrix, int i, int j, String ans,
Vector<String> paths) {
    if(i == 0 || j == 0) {
        ans = "(0,0)-> " + ans;
        paths.add(ans);
        return;
    }
    int fromAbove = matrix[i-1][j].entry + matrix[i-1][j].goDown;
    int fromLeft = matrix[i][j-1].entry + matrix[i][j-1].goRight;
    String directionLeft = "("+i+","+(j-1)+")-> ";
    String directionAbove = "(" + (i - 1) + "," + j + ")-> ";
    if(fromAbove > fromLeft) {
        getPaths(matrix,i,j-1, directionLeft + ans, paths);
    } else if(fromAbove < fromLeft) {
        getPaths(matrix, i - 1, j, directionAbove + ans, paths);
    } else {
        getPaths(matrix,i,j-1, directionLeft + ans, paths);
        getPaths(matrix, i - 1, j, directionAbove + ans, paths);
    }
}
```