Collect maximum coins before hitting a dead end

Given a character matrix where every cell has one of the following values.

```
'C' -->  This cell has coin

'#' -->  This cell is a blocking cell.
         We can not go anywhere from this.

'E' -->  This cell is empty. We don't get
         a coin, but we can move from here.
```

Initial position is cell (0, 0) and initial direction is right.

Following are rules for movements across cells.

If face is Right, then we can move to below cells

1. Move one step ahead, i.e., cell (i, j+1) and direction remains right.
2. Move one step down and face left, i.e., cell (i+1, j) and direction becomes left.

If face is Left, then we can move to below cells

1. Move one step ahead, i.e., cell (i, j-1) and direction remains left.
2. Move one step down and face right, i.e., cell (i+1, j) and direction becomes right.

Final position can be anywhere and final direction can also be anything. The target is to collect maximum coins.

Example:

Input:

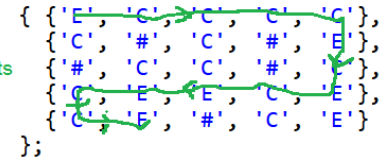
```
char arr[R][C] = { {'E', 'C', 'C', 'C', 'C'},
                   {'C', '#', 'C', '#', 'E'},
                   {'#', 'C', 'C', '#', 'C'},
                   {'C', 'E', 'E', 'C', 'E'},
                   {'C', 'E', '#', 'C', 'E'}
                 };
```

Output:

Maximum number of collected coins is 8

Explanation:

Highlighted is path to collect maximum 8 coints



We strongly recommend you to minimize your browser and try this yourself first.

The above problem can be recursively defined as below:

```
maxCoins(i, j, d):  Maximum number of coins that can be
                    collected if we begin at cell (i, j)
                    and direction d.
                    d can be either 0 (left) or 1 (right)

   // If this is a blocking cell, return 0. isValid() checks
   // if i and j are valid row and column indexes.
   If (arr[i][j] == '#' or isValid(i, j) == false)
       return 0

   // Initialize result
   If (arr[i][j] == 'C')
       result = 1;
   Else
       result = 0;

   If (d == 0) // Left direction
       return result +  max(maxCoins(i+1, j, 1),  // Down
                            maxCoins(i, j-1, 0)); // Ahead in left

   If (d == 1) // Right direction
       return result +  max(maxCoins(i+1, j, 1),  // Down
                            maxCoins(i, j+1, 0)); // Ahead in right
```

Below is C++ implementation of above recursive algorithm.

```
// A Naive Recursive C++ program to find maximum number of coins
// that can be collected before hitting a dead end
#include<bits/stdc++.h>
using namespace std;
#define R 5
#define C 5

// to check whether current cell is out of the grid or not
bool isValid(int i, int j)
{
    return (i >=0 && i < R && j >=0 && j < C);
}

// dir = 0 for left, dir = 1 for facing right.  This function returns
// number of maximum coins that can be collected starting from (i, j).
int maxCoinsRec(char arr[R][C],  int i, int j, int dir)
{
    // If this is a invalid cell or if cell is a blocking cell
    if (isValid(i,j) == false || arr[i][j] == '#')
        return 0;

    // Check if this cell contains the coin 'C' or if its empty 'E'.
    int result = (arr[i][j] == 'C')? 1: 0;

    // Get the maximum of two cases when you are facing right in this cell
    if (dir == 1) // Direction is right
       return result + max(maxCoinsRec(arr, i+1, j, 0),     // Down
                           maxCoinsRec(arr, i, j+1, 1));  // Ahead in right

    // Direction is left
    // Get the maximum of two cases when you are facing left in this cell
     return  result + max(maxCoinsRec(arr, i+1, j, 1),    // Down
                          maxCoinsRec(arr, i, j-1, 0));  // Ahead in left
}

// Driver program to test above function
int main()
{
    char arr[R][C] = { {'E', 'C', 'C', 'C', 'C'},
                       {'C', '#', 'C', '#', 'E'},
                       {'#', 'C', 'C', '#', 'C'},
                       {'C', 'E', 'E', 'C', 'E'},
                       {'C', 'E', '#', 'C', 'E'}
                     };

   // As per the question initial cell is (0, 0) and direction is
    // right
    cout << "Maximum number of collected coins is "
         << maxCoinsRec(arr, 0, 0, 1);

    return 0;
}
```

Output:

```
Maximum number of collected coins is 8
```

The time complexity of above solution recursive is exponential. We can solve this problem in Polynomial Time using Dynamic Programming. The idea is to use a 3 dimensional table dp[R][C][k] where R is number of rows, C is number of columns and d is direction. Below is Dynamic Programming based C++ implementation.

```
// A Dynamic Programming based C++ program to find maximum
// number of coins that can be collected before hitting a
// dead end
#include<bits/stdc++.h>
using namespace std;
#define R 5
#define C 5

// to check whether current cell is out of the grid or not
bool isValid(int i, int j)
```

```
bool isValid(int i, int j)
{
    return (i >=0 && i < R && j >=0 && j < C);
}

// dir = 0 for left, dir = 1 for right.  This function returns
// number of maximum coins that can be collected starting from
// (i, j).
int maxCoinsUtil(char arr[R][C],  int i, int j, int dir,
                 int dp[R][C][2])
{
    // If this is a invalid cell or if cell is a blocking cell
    if (isValid(i,j) == false || arr[i][j] == '#')
        return 0;

    // If this subproblem is already solved than return the
    // already evaluated answer.
    if (dp[i][j][dir] != -1)
       return dp[i][j][dir];

    // Check if this cell contains the coin 'C' or if its 'E'.
    dp[i][j][dir] = (arr[i][j] == 'C')? 1: 0;

    // Get the maximum of two cases when you are facing right
    // in this cell
    if (dir == 1) // Direction is right
       dp[i][j][dir] += max(maxCoinsUtil(arr, i+1, j, 0, dp), // Down
                            maxCoinsUtil(arr, i, j+1, 1, dp)); // Ahead in rught

    // Get the maximum of two cases when you are facing left
    // in this cell
    if (dir == 0) // Direction is left
       dp[i][j][dir] += max(maxCoinsUtil(arr, i+1, j, 1, dp),  // Down
                            maxCoinsUtil(arr, i, j-1, 0, dp)); // Ahead in left

    // return the answer
    return dp[i][j][dir];
}

// This function mainly creates a lookup table and calls
// maxCoinsUtil()
int maxCoins(char arr[R][C])
{
    // Create lookup table and initialize all values as -1
    int dp[R][C][2];
    memset(dp, -1, sizeof dp);

    // As per the question initial cell is (0, 0) and direction
    // is right
    return maxCoinsUtil(arr, 0, 0, 1, dp);
}

// Driver program to test above function
int main()
{
    char arr[R][C] = { {'E', 'C', 'C', 'C', 'C'},
                       {'C', '#', 'C', '#', 'E'},
                       {'#', 'C', 'C', '#', 'C'},
                       {'C', 'E', 'E', 'C', 'E'},
                       {'C', 'E', '#', 'C', 'E'}
                     };


    cout << "Maximum number of collected coins is "
         << maxCoins(arr);

    return 0;
}
```

Output:

```
Maximum number of collected coins is 8
```

Time Complexity of above solution is O(R x C x d). Since d is 2, time complexity can be written as O(R x C).