给出策略打印旋转矩阵

问题难点

实际上打印旋转矩阵就是要找到如何旋转的规律关系。首先从左下角开始打印，起始的横坐标为当前大矩阵的最后一行，起始纵坐标为当前大矩阵的第一列，并且考虑到当前大矩阵的阶数n三者综合考虑之间的关系。

算法思路

i为当前大矩阵的第一个元素的横坐标，j为当前大矩阵的第一个元素的纵坐标。

要得到当前大矩阵的最后一行的横坐标，则u = i+n-1

要得到当前大矩阵的第一列，则v = j;

找到第一次迭代的起始位置，环绕一圈的规律就很容易得到。

进行下一次迭代前，当前大矩阵的第一个元素的坐标将发生改变，即向右下角移动一次

所以i = i + 1，j = j + 1

但是大矩阵的阶数会因为上层和下层已经被填充，所以实际上n = n - 2

所以第二次迭代为fill(i + 1 , j + 1, n - 2)

代码

```cpp
#include <stdio.h>
#include <iostream>
using namespace std;

int a[100][100]; // 定义二维数组存储矩阵
int m = 1; // 定义变量m表示当前填充的数字

void fill(int i, int j, int n)
{
    if (n <= 0)
        return; // 递归出口，n小于等于0时返回
    if (n == 1)
    {
        a[i][j] = m;
        return;// 填充最后一个数字
    }
    int u = i + n - 1;
    int v = j;
    for (int k = 0; k < n; k++)// 从左到右填充底部
        a[u][v++] = m++;
    v--;
    u--;
    for (int k = 0; k < n - 1; k++)// 从下到上填充右部
        a[u--][v] = m++;
    u++;
    v--;
    for (int k = 0; k < n - 1; k++)// 从右到左填充上部
        a[u][v--] = m++;
    v++;
    u++;
    for (int k = 0; k < n - 2; k++)// 从上到下填充左部
        a[u++][v] = m++;
    fill(i + 1, j + 1, n - 2); //递归
}

int main()
{
    int n;
    cout << "请输入矩阵阶数" << endl;
    cin >> n;
    fill(0, 0, n);
    for (int i = 0; i<n; i++)
    {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << "\t";
        cout<<endl;
    }
    return 0;
}
```

实现顺时针旋转矩阵90°

问题分析

将矩阵顺时针旋转90°后，第一列从下到上变为第一行从左到右，第二列从下到上变为第二行从左到右....

由此可以看出其坐标其中的关系，找到此关系就可以依次打印出旋转矩阵，比较简单。

代码

```
#include<iostream>
#include<vector>
using namespace std;
int b[200][200];
void rotate(vector<vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();
        for (int i = n - 1, k = 0; i >= 0; i--, k++) {//for循环用b数组保存旋转后的矩阵
                for (int j = 0; j < m; j++) {
                        b[j][i] = matrix[k][j];
                }
        }
        for (int i = 0; i < m; i++) {//输出旋转后的矩阵
                for (int j = 0; j < n; j++) {
                        cout << b[i][j];
                        if (j != n - 1)
                                cout << " ";
                }
                cout << endl;
        }
}
int main() {
        vector<vector<int>> matrix = {
                {8, 3, 5,1},
                {1, 7, 1,1},
                {4, 9, 9,9},
                {2, 2, 1,1}
        };
        rotate(matrix);
        return 0;

}
```

实现“之”字形打印矩阵

问题分析

从第一个元素开始打印，若要实现“之”字形，则要找到对应元素的坐标关系。

从左下到右上的对角线的元素中，相邻元素间横坐标纵坐标都差1

每一次打印一对角线元素中，起始和终止元素都是矩阵的边界元素。右上的起始元素的位置依次向右，达到对称点出则向下。

左下的起始元素依次向下，达到对称点后向右。

代码

```cpp
#include<iostream>
#include<vector>

using namespace std;

void printArray(vector<vector<int>>& arr, int row1, int col1, int row2, int col2, bool toUp)
{
    if (toUp)
        while (row1 <= row2) cout << arr[row2--][col2++] << " ";// 向右上角打印，则行坐标依次减小，
    else
        while (row1 <= row2) cout << arr[row1++][col1--] << " ";// 向左下角打印，则行坐标依次增大，
}

int main()
{
    vector<vector<int>> matrix = {
        {8, 3, 5,1},
        {1, 7, 1,1},
        {4, 9, 9,9},
        {2, 2, 1,1}
    };
    int m = matrix.size();      //得到矩阵行数
    int n = matrix[0].size();   //得到矩阵列数
    int row1 = 0, col1 = 0;             //row1，col1为向上或者向下打印时最顶部的元素的坐标
    int row2 = 0, col2 = 0;             //row2,col2为向上或向下打印时最底部的元素的坐标
    bool toUp = true;                   //起始打印为向右上（因为只有一个元素所以看不出）
    while (row1 < n)
    {
        printArray(matrix, row1, col1, row2, col2, toUp);
        col1 == m - 1 ? row1++ : col1++;        //打印是对称的，所以顶部的元素先向右，达到对称点后向
        row2 == n - 1 ? col2++ : row2++;        //打印是对称的，所以顶部的元素先向下，达到对称点后向
        toUp = !toUp;
    }
    return 0;
}
```

给出策略返回一个m*n矩阵中从【1,1】到【m,n】的和最小的一条路径。

问题：

这是一个经典的动态规划问题，可以用动态规划算法来解决。

算法：

- 首先定义一个二维数组dp，其中dp[i] [j]表示从起点[1] [1]到达位置[i][j]时的最小路径和。

- 初始化dp数组，dp[1] [1]=matrix[1] [1]（matrix是给出的m*n矩阵）。
- 对于dp的第一行和第一列，因为它们只有一个方向可以走，所以它们到起点[1] [1]的路径和可以直接计算得到。因此，我们可以分别计算dp[1] [j]和dp[i] [1]，其中dp[1] [j]=dp[1] [j-1]+matrix[1] [j]，dp[i] [1]=dp[i-1] [1]+matrix[i] [1]。
- 对于其他位置[i] [j]，因为可以从上方或左方到达，所以可以选择其中路径和最小的那个方向。因此，我们可以使用递推式dp[i] [j]=min(dp[i-1] [j], dp[i] [j-1])+matrix[i] [j]来计算。其中，min()函数用来选择路径和最小的方向。
- 最后，dp[m] [n]就是从起点[1] [1]到终点[m] [n]的最小路径和。我们可以倒推出最短路径，具体方法是从dp[m] [n]开始，向左或向上选择路径和更小的方向一直走到起点[1] [1]。

代码：

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<pair<int, int>> minPathSum(vector<vector<int>>& matrix) {
    int m = matrix.size();
    int n = matrix[0].size();
    vector<vector<int>> dp(m, vector<int>(n)); //建立dp动态矩阵
    dp[0][0] = matrix[0][0];

    // 初始化第一行和第一列
    for (int j = 1; j < n; j++) {
        dp[0][j] = dp[0][j - 1] + matrix[0][j];
    }
    for (int i = 1; i < m; i++) {
        dp[i][0] = dp[i - 1][0] + matrix[i][0];
    }

    // 计算其他位置的最小路径和
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + matrix[i][j];
        }
    }

    // 回溯出最短路径
    vector<pair<int, int>> path;
    int i = m - 1, j = n - 1;
    path.emplace_back(m - 1, n - 1);
    while (i > 0 || j > 0) {
        if (i == 0) {
            path.emplace_back(0, j - 1);
            j--;
        }
        else if (j == 0) {
            path.emplace_back(i - 1, 0);
            i--;
        }
        else {
            if (dp[i - 1][j] < dp[i][j - 1]) {
                path.emplace_back(i - 1, j);
                i--;
            }
            else {
                path.emplace_back(i, j - 1);
                j--;
            }
        }
```

```
    }
    reverse(path.begin(), path.end());
    return path;
}

int main() {
    vector<vector<int>> matrix = {
        {1, 1, 1,1},
        {1, 1, 1,1},
        {4, 9, 9,9},
        {2, 2, 1,1}
    };
    vector<pair<int, int>> path = minPathSum(matrix);
    for (auto& p : path) {
        cout << "[" << p.first << "," << p.second << "] ";
    }
    cout << endl;
    return 0;
}
```