调研学习Voronoi图及相关算法，并给出至少一个应用。

基本概念：

Voronoi图，又叫泰森多边形或Dirichlet图，它是由一组由连接两邻点直线的垂直平分线组成的连续多边形组成。N个在平面上有区别的点，按照最邻近原则划分平面;每个点与它的最近邻区域相关联。Delaunay三角形是由与相邻Voronoi多边形共享一条边的相关点连接而成的三角形。Delaunay三角形的外接圆圆心是与三角形相关的Voronoi多边形的一个顶点。Voronoi三角形是Delaunay图的偶图

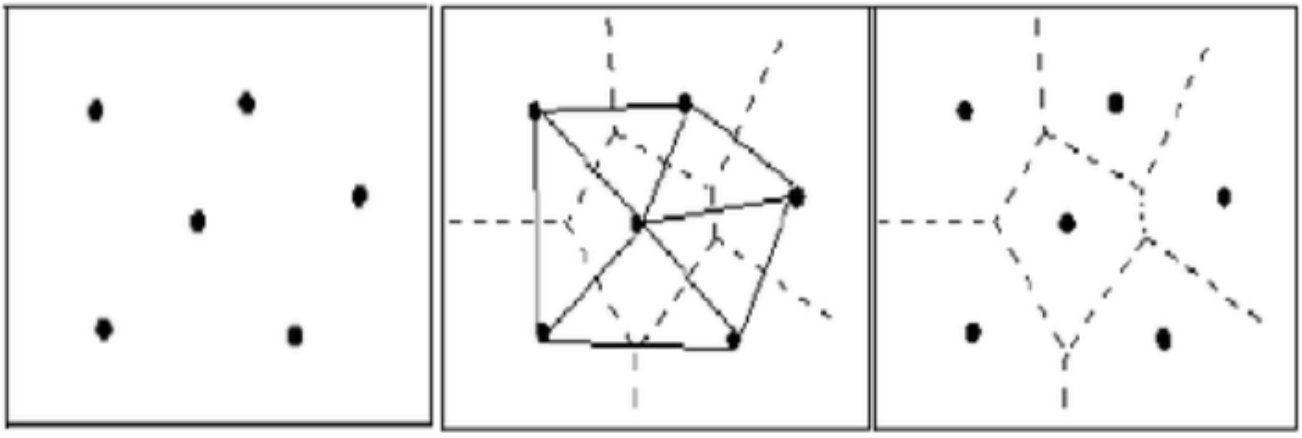
特点

- 每个V多边形内有一个生成元；
- 每个V多边形内点到该生成元距离短于到其它生成元距离；
- 多边形边界上的点到生成此边界的生成元距离相等；
- 邻接图形的Voronoi多边形界线以原邻接界线作为子集。

Voronoi图的生成

生成V图的方法很多，常见的有分治法、扫描线算法和Delaunay三角剖分算法。

这里介绍的是Delaunay三角剖分算法。主要是指生成Voronoi图时先生成其对偶元Delaunay三角网，再找出三角网每一三角形的外接圆圆心，最后连接相邻三角形的外接圆圆心，形成以每一三角形顶点为生成元的多边形网。



建立Voronoi图算法的关键是对离散数据点合理地连成三角网，即构建Delaunay三角网。
建立Voronoi图的步骤为：

- 离散点自动构建三角网，即构建Delaunay三角网。对离散点和形成的三角形编号，记录每个三角形是由哪三个离散点构成的。
- 计算每个三角形的外接圆圆心，并记录之。
- 遍历三角形链表，寻找与当前三角形pTri三边共边的相邻三角形TriA，TriB和TriC。
- 如果找到，则把寻找到的三角形的外心与pTri的外心连接，存入维诺边链表中。如果找不到，则求出最外边的中垂线射线存入维诺边链表中。
- 遍历结束，所有维诺边被找到，根据边画出维诺图。

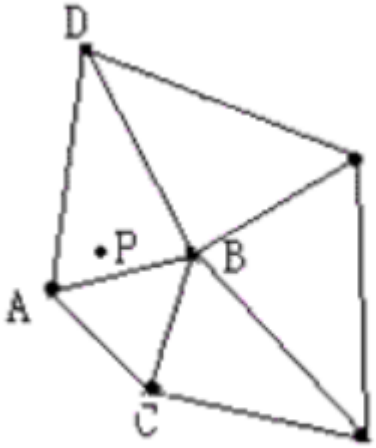
那么如何生成Delaunay三角网呢?

首先了解一下Delaunay三角网的特性:

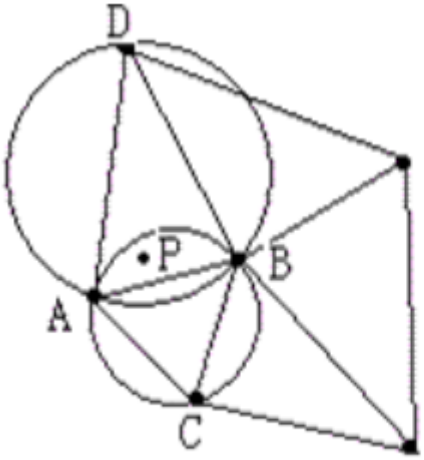
- 空圆性，任一三角形外接圆内部不包含其他点。
- 最接近：以最近临的三点形成三角形，且各线段(三角形的边)皆不相交。
- 唯一性：不论从区域何处开始构建，最终都将得到一致的结果。
- 最优性：任意两个相邻三角形形成的凸四边形的对角线如果可以互换的话，那么两个三角形六个内角中最小的角度不会变大。
- 最规则：如果将三角网中的每个三角形的最小角进行升序排列，则Delaunay三角网的排列得到的数值最大。
- 区域性：新增、删除、移动某一个顶点时只会影响临近的三角形。
- 具有凸多边形的外壳：三角网最外层的边界形成一个凸多边形的外壳。

这里采用Bowyer-Watson算法得到Delaunay三角网，算法的基本步骤是:

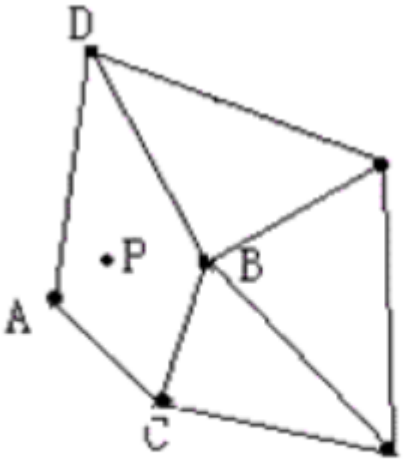
- 构造一个超级三角形，包含所有散点，放入三角形链表。
- 将点集中的散点依次插入，在三角形链表中找出其外接圆包含
  插入点的三角形（称为该点的影响三角形），删除影响三角形的公共边，将插入点同影响三角形的全部顶点连接起来，从而完成一个点在Delaunay三角形链表中的插入。
- 根据优化准则对局部新形成的三角形进行优化。将形成的三角形放入Delaunay三角形链表。
- 循环执行上述第2步，直到所有散点插入完毕。



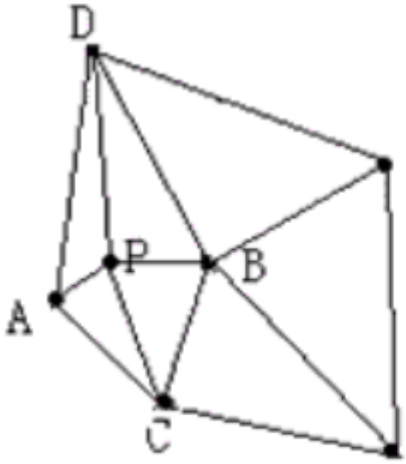
(a)插入新结点P



(b)决定如何连接P与其它顶点



(c)删除边AB



(d)形成三角形

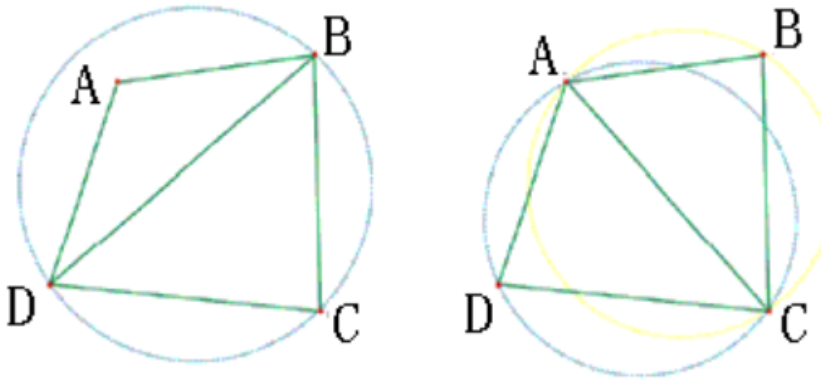
步骤3的局部优化的准则指的是：

1.对新形成的三角形进行优化，将两个具有共同边的三角形合成一个多边形。

2.以最大空圆准则作检查，看其第四个顶点是否在三角形的外接圆之内。

3.如果在，修正对角线即将对角线对调，即完成局部优化过程的处理。

LOP (Local Optimization Procedure)处理过程如下图所示：



应用举例：

当地形地貌数据进行空间分析时，经常使用Voronoi图进行地形地貌分析。Voronoi图中的每个Voronoi单元代表一个离散点的周围地形地貌特征（如高程、坡度、曲率等）。

通过对Voronoi图的计算和分析，可以获得许多有用的地形地貌信息，如地形起伏程度、地形粗糙度、坡度、曲率等。这些信息对于地质勘探、土地利用规划、环境保护和灾害防治等领域都有着重要的应用价值。

举个例子，如果想要对一个山区进行水土保持规划，需要了解该区域的地形特征。通过构建该区域的Voronoi图，可以得到每个离散点周围的地形特征，从而了解该区域的地形变化、地貌特征以及潜在的水土流失风险等信息。在此基础上，可以制定出相应的水土保持规划措施，如种植防护林带、修建护坡等，从而有效保护该区域的生态环境和土地资源。

前面已经学习在一维序列中某元素的查找算法，现若查找是在一个二维矩阵中进行，并且矩阵中行列均为升序排列，请给出相应查找算法并进行分析。

问题分析：

实际上这个问题就是查找问题，不过从一维变成了二维。一些算法还是可以沿用一维时的算法。

算法一

使用二分查找来解决，其时间复杂度为 O(log n)，其中 n 是矩阵中元素的个数。

具体来说，我们可以按照行的顺序依次扫描矩阵的每一行。对于每一行，我们可以使用二分查找在该行中查找目标元素。如果找到目标元素，就返回其位置；否则，我们将查找范围缩小到该行左上角和右下角之间的矩形区域内，并继续进行二分查找。

其算法步骤如下：

- 初始化行下标 start 为 0，列下标 end 为矩阵的总元素数减 1。
- 重复以下步骤直到找到目标元素或搜索区域为空： a. 计算中间行 mid = (start + end) / 2。
- 得到中间元素的行row = mid/n
- 得到中间元素的列col = mid%n
- 如果目标元素小于matrix[row] [col]，那么将搜索区域缩小到左上角到 (start, mid-1) 的矩形内。
- 如果目标元素大于matrix[row] [col]，那么将搜索区域缩小到 (mid+1, end) 到右下角的矩形内。
- 否则，在第 mid 行中进行二分查找，如果找到目标元素，返回其位置；否则，将搜索区域缩小到该行左上角和右下角之间的矩形区域内。
- 如果没有找到目标元素，返回 -1。

代码

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

using namespace std;

int searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size(); //行数
    if (m == 0) {
        return -1;
    }
    int n = matrix[0].size(); //列数
    int start = 0;
    int end = m * n - 1;
    while (start <= end) {
        int mid = (start + end) / 2;
        int row = mid / n; //得到中间元素的行
        int col = mid % n; //得到中间元素的列
        if (matrix[row][col] == target) {
            return mid;
        }
        if (matrix[row][col] < target) {
            start = mid + 1;
        }
        else {
            end = mid - 1;
        }
    }
    return -1;
}

int main() {
    vector<vector<int>> matrix = { {1, 3, 5, 7}, {10, 11, 16, 20}, {23, 30, 34, 50} };
    int target = 10;
    int index = searchMatrix(matrix, target);
    if (index == -1) {
        cout << "找不到该元素" << endl;
    }
    else {
        cout << "元素的位置为: " << index << endl;
    }
    return 0;
}
```

运行结果为:

算法二

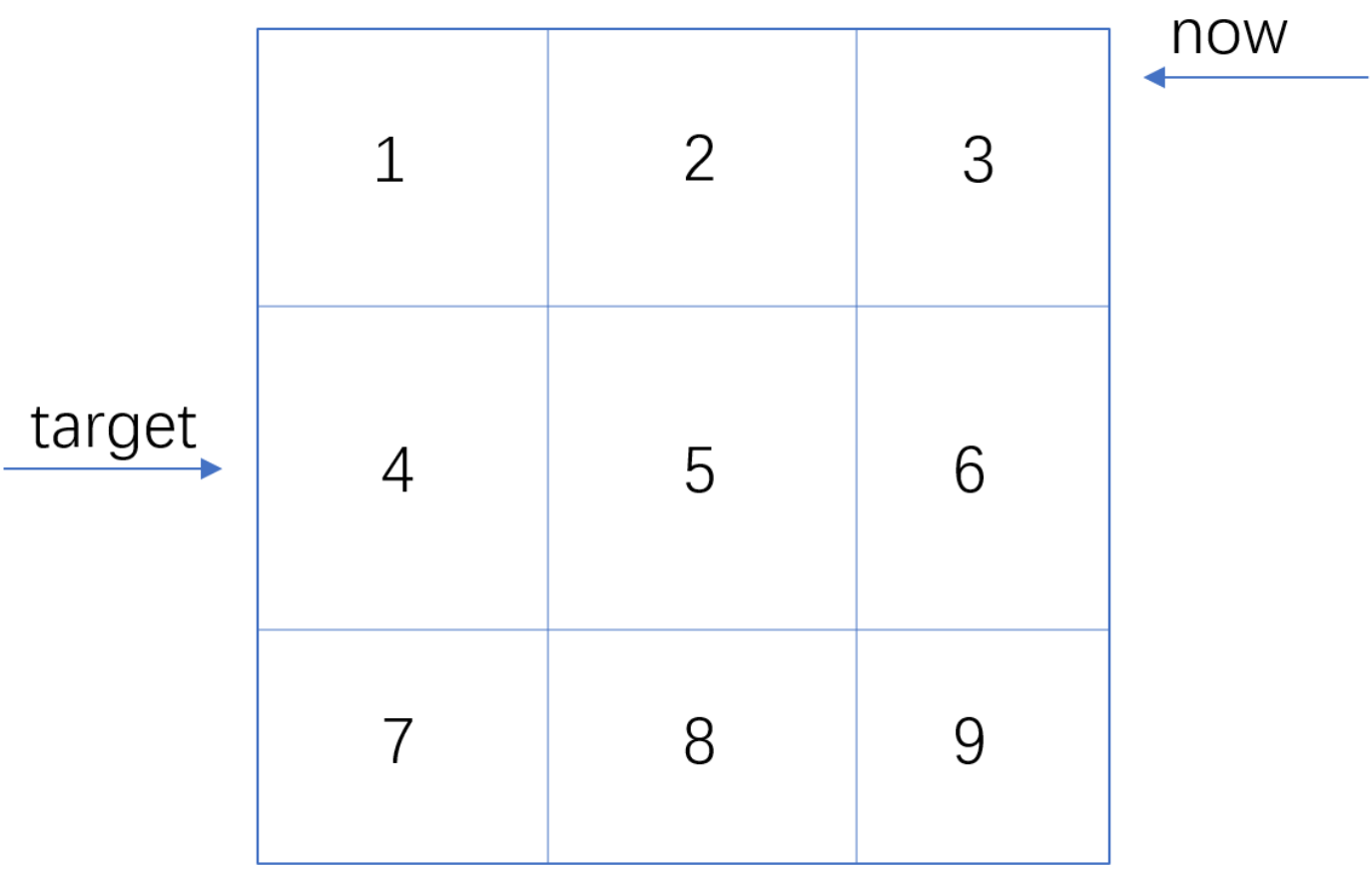
还有一种更简单的算法可以快速缩小查找范围。

从矩阵的右上角或左下角开始查找。这种方法的时间复杂度为 O(m+n)，其中 m 和 n 分别是矩阵的行数和列数。

其算法步骤为：

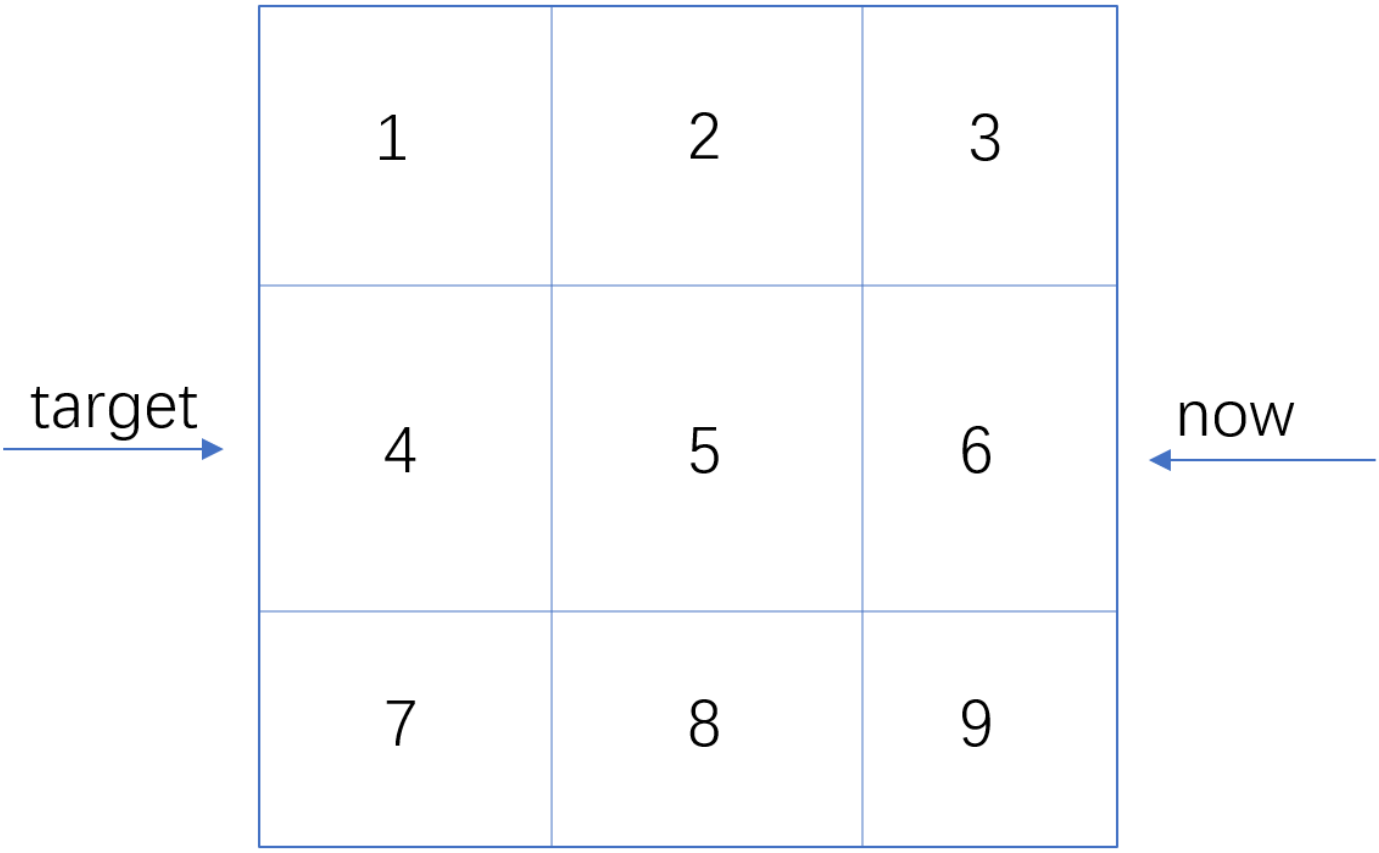
- 从右上角开始匹配，如果该元素大于目标元素
- 则将匹配向左移动一列
- 反之则将匹配向下移动一行

举例：



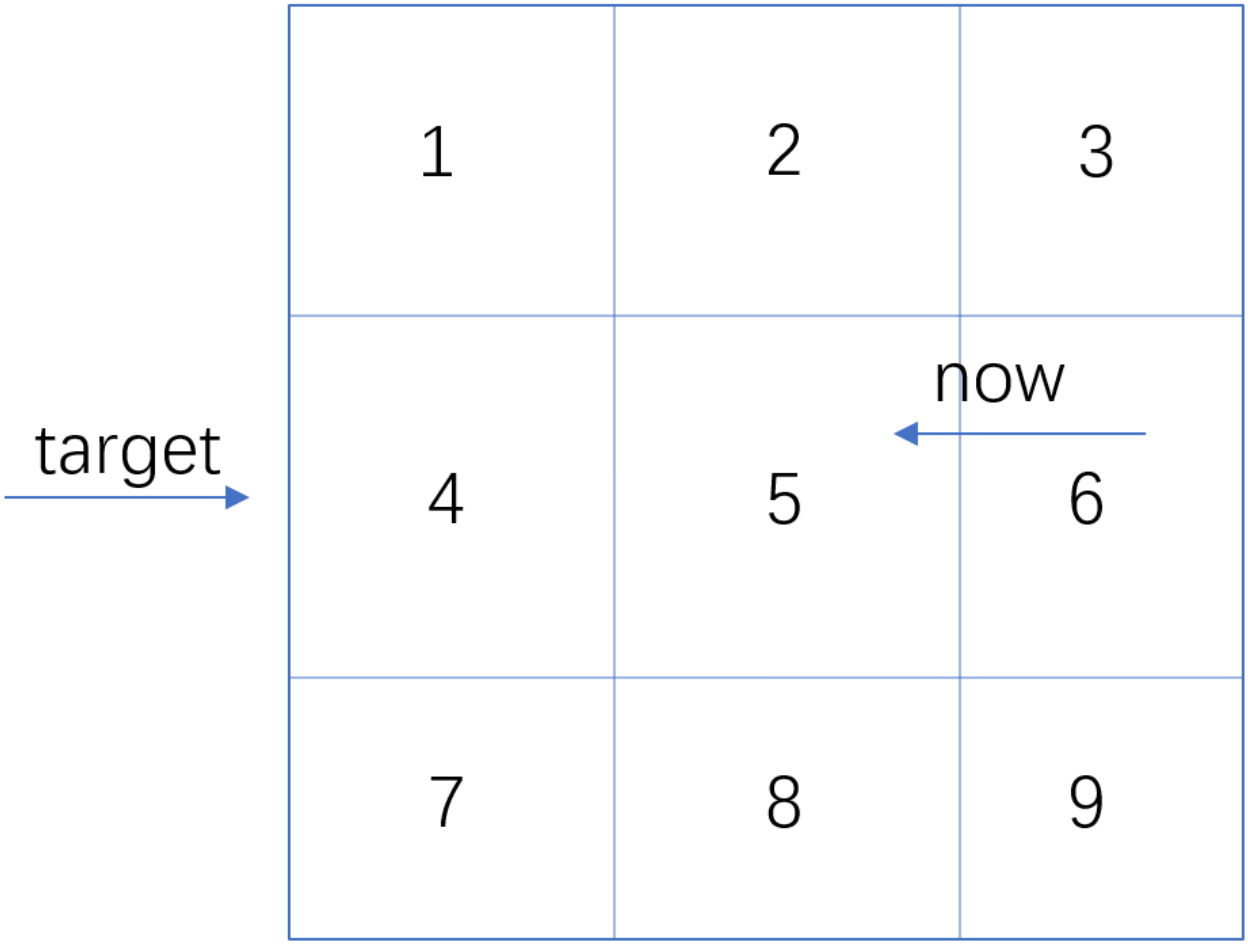
3小于4

则下移一位



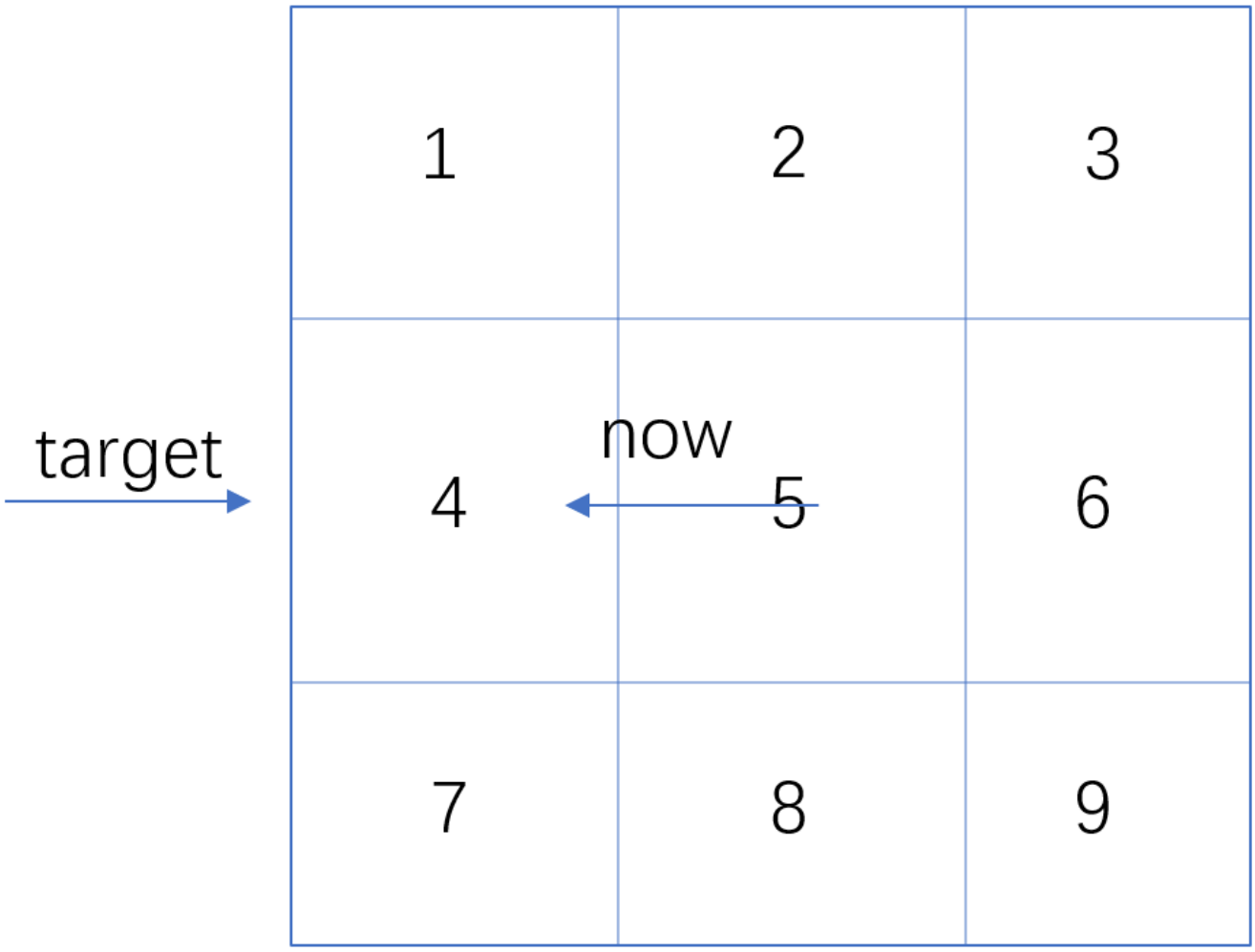
6大于4

则左移一位



5大于4

则左移一位



返回1*3+0 = 3

代码

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

using namespace std;

int searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size();
    if (m == 0) {
        return -1;
    }
    int n = matrix[0].size();
    int row = 0;
    int col = n - 1;
    while (row < m && col >= 0) {
        if (matrix[row][col] == target) {
            return row * n + col;
        }
        else if (matrix[row][col] < target) {
            row++;
        }
        else {
            col--;
        }
    }
    return -1;
}


int main() {
    vector<vector<int>> matrix = { {1, 3, 5, 7}, {10, 11, 16, 20}, {23, 30, 34, 50} };
    int target = 10;
    int index = searchMatrix(matrix, target);
    if (index == -1) {
        cout << "找不到该元素" << endl;
    }
    else {
        cout << "元素的位置为: " << index << endl;
    }
    return 0;
}
```

运行结果为:


Microsoft Visual Studio 调试控制台
元素的位置为: 4
C:\Users\lenovo\Desktop\Project1\x64\Debug\Pro
要在调试停止时自动关闭控制台，请启用“工具”->
按任意键关闭此窗口. . .