Searching a 2D Sorted Matrix Part II

October 8, 2010 by 1337c0d3r 34 Replies

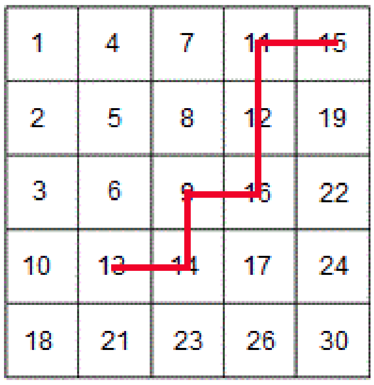
Write an efficient algorithm that searches for a value in an *n* x *m* table (two-dimensional array). This table is sorted along the rows and columns — that is,

Table[i][j] ≤ Table[i][j + 1],
Table[i][j] ≤ Table[i + 1][j]

Note:
This is Part II of the article: Searching a 2D Sorted Matrix. Please read Part I for more background information.

Step-wise Linear Search:
We call this the *Step-wise Linear Search* method. Similar to *Diagonal Binary Search*, we begin with the upper right corner (or the bottom left corner). Instead of traversing diagonally each step, we traverse one step to the left or bottom. For example, the picture below shows the traversed path (the red line) when we search for **13**.

Essentially, each step we are able to eliminate either a row or a column. The worst case scenario is where it ended up in the opposite corner of the matrix, which takes at most 2*n* steps. Therefore, this algorithm runs in O(*n*) time, which is better than previous approaches.



1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

An example showing the traversed path using step-wise linear search (colored in red) when the target element is **13**.

Below is the code and it is simple and straight to the point. You should not make any careless mistake during the interview.

```
bool stepWise(int mat[][N_MAX], int N, int target,
              int &row, int &col) {
  if (target < mat[0][0] || target > mat[N-1][N-1]) return false;
  row = 0;
  col = N-1;
  while (row <= N-1 && col >= 0) {
    if (mat[row][col] < target)
      row++;
    else if (mat[row][col] > target)
      col--;
    else
      return true;
  }
  return false;
}
```

This is probably the answer that most interviewers would be looking for. But we will not stop here. Let us continue exploring some more interesting solutions.

Quad Partition:

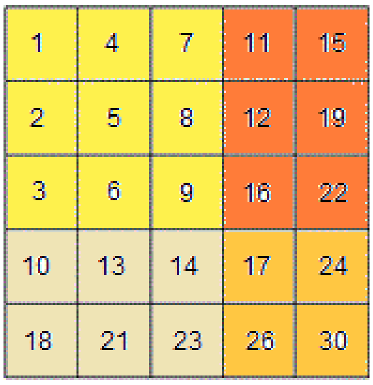
Did you realize that this problem is actually solvable using a divide and conquer approach? I bet you did!

First, we make an observation that the center element always partition the matrix into four smaller matrices. For example, the center element **9** partitions the matrix into four matrices as shown in the picture below. Since the four smaller matrices are also sorted both row and column-wise, the problem can naturally be divided into four sub-problems.

If you notice carefully, we are always able to eliminate one of the four sub-problems in each step. Assume our target is **21**, which is greater than the center element **9**. We can eliminate the upper left quadrant instantly, because all elements in that quadrant are always less than or equal to **9**. Now assume our target is **6**, which is less than **9**.

Similarly, we eliminate the bottom right quadrant from consideration, because elements in that quadrant must all be greater than **9**. Please note however, we still need to search the upper right and bottom left quadrant, even though the example below seems to show all elements in the two mentioned quadrants are greater than **9**.

Of course, if the center element is our target element, we have found the target and stop searching. If not, we proceed by searching the rest of three quadrants.



1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

The center element **9** partitions the matrix into four smaller quadrants (shown as four different colors).

What's the complexity of the *Quad Partition* method? As it turns out, the run time complexity could be written directly as a recurrence relation:

```
T(n) = 3T(n/2) + c,

where n is the dimension of the matrix.
```

We add a constant *c* because each step we do a comparison between the target element and the center element, which takes some constant time.

We need to solve the above equation to obtain the complexity. This is where most confusion comes in. If you have taken advanced algorithm course, you could solve it using the Master's theorem, but you *don't really need to*. You could just expand the recurrence relation directly to solve it.

```
T(n) = 3T(n/2) + c,
     = 3 [ 3T(n/4) + c ] + c
     = 3 [ 3 [ 3T(n/8)  + c ] + c ] + c
     = 3^k T(n/2^k) + c (3^k - 1)/2
     = 3^k ( T(n/2^k) + c ) - c/2



Setting k = lg n,
T(n)  = 3^(lg n) ( T(1) + c ) - c/2
      = O(3^(lg n))
      = O(n^(lg 3))           <== 3^(lg n) = n^(lg 3)
      = O(n^1.58)
```

Below is the code for the *Quad Partition* method. *l* and *u* represents the upper left corner, while *r* and *d* represents the bottom right corner of the matrix. Be very careful of corner cases. Please note that the code below checks for when *l* equals *r* (left = right) and *u* equals *d* (up = down) (ie, the matrix has only one element). If this only element differs from the target, the function must return false. If you omit this condition, then the code below never terminates, which in other word translates to: *You never double check your code*, and it is Hasta la vista, baby from your interviewer.

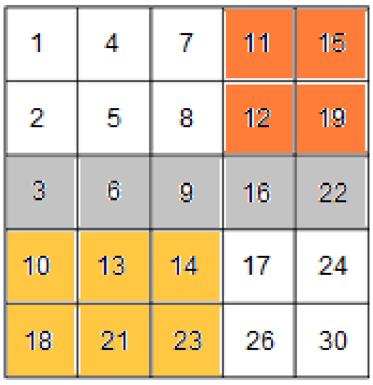
```
bool quadPart(int mat[][N_MAX], int M, int N, int target, int l, int u, int r, int d, int &ta
  if (l > r || u > d) return false;
  if (target < mat[u][l] || target > mat[d][r]) return false;
  int col = l + (r-l)/2;
  int row = u + (d-u)/2;
  if (mat[row][col] == target) {
    targetRow = row;
    targetCol = col;
    return true;
  } else if (l == r && u == d) {
    return false;
  }
  if (mat[row][col] > target) {
    return quadPart(mat, M, N, target, col+1, u, r, row, targetRow, targetCol) ||
           quadPart(mat, M, N, target, l, row+1, col, d, targetRow, targetCol) ||
           quadPart(mat, M, N, target, l, u, col, row, targetRow, targetCol);
  } else {
    return quadPart(mat, M, N, target, col+1, u, r, row, targetRow, targetCol) ||
           quadPart(mat, M, N, target, l, row+1, col, d, targetRow, targetCol) ||
           quadPart(mat, M, N, target, col+1, row+1, r, d, targetRow, targetCol);
  }
}

bool quadPart(int mat[][N_MAX], int N, int target, int &row, int &col) {
  return quadPart(mat, N, N, target, 0, 0, N-1, N-1, row, col);
}
```

Binary Partition:

We can even reduce the number of sub-problems from three to only two using a method we called *Binary Partition*. This time we traverse along either the middle row, middle column, or diagonally (as shown in highlighted gray cells in images *a*), *b*), and *c*) below). As we traverse, we find the point such that the target element *s* satisfies the following condition:

$a_i < s < a_{i+1}$,

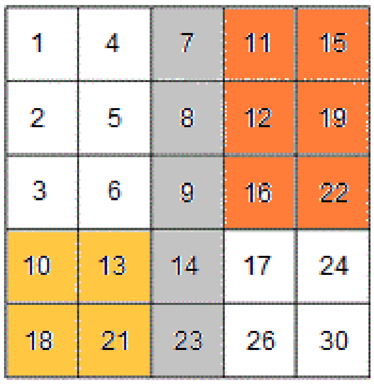
where a_i is the i^{th} traversed cell.



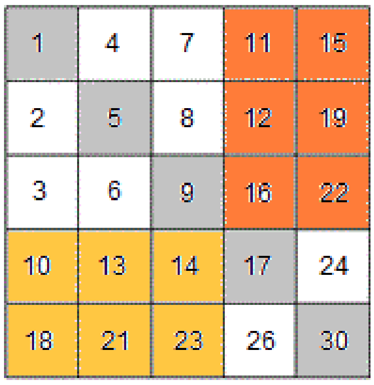
1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

a) Row-wise binary partition. The highlighted gray cells represents the traversed row (the middle row). The target **10** is found between **9** and

16.



b) Column-wise binary partition. The highlighted gray cells represents the traversed column (the middle column). The target **10** is found between **9** and **14**.



c) Diagonal-based binary partition. The highlighted gray cells represents the traversed diagonal. The target **10** is found between **9** and **17**. Please note that diagonal-based binary partition would fail in a non-square matrix (for the above example, it will not work in the two sub-matrices because they are non-square matrices).

If the target element equals one of the traversed cells, we immediately return the element as found. Otherwise we partition the matrix into two sub-matrices following the partition point we found. As it turns out, we need *cn* time (linear time) to find such partition point, since we are essentially performing a linear search. Therefore, the complexity could be written as the following recurrence relation: (Note: I omitted the proof, as it is left as an exercise to the reader.)

```
T(n) = 2T(n/2) + cn
     = O(n lg n)
```

The *Binary Partition* algorithm runs in *O*(*n* lg *n*) time. You might expect its complexity to be lower than *Quad Partition*, since it has only two sub-problems (instead of three) to solve. The reason of the higher order complexity is due to the extra *O*(*n*) time doing a linear search for the partition point where $a_i < s < a_{i+1}$.

Please note that the matrix is not necessarily divided into two equal-sized sub-matrices. One of the matrix could be bigger than the other one, or in the extreme case, the other matrix could be of size zero. Here, we have made an assumption that each partition step divides the matrix into two equal-sized sub-matrices, just for the sake of complexity analysis.

Below is the code for the *Binary Partition* method. The code below chooses the middle column as the place to search for the partition point.

```
bool binPart(int mat[][N_MAX], int M, int N, int target, int l, int u, int r, int d, int &tar
  if (l > r || u > d) return false;
  if (target < mat[u][l] || target > mat[d][r]) return false;
  int mid = l + (r-l)/2;
  int row = u;
  while (row <= d && mat[row][mid] <= target) {
    if (mat[row][mid] == target) {
      targetRow = row;
      targetCol = mid;
      return true;
    }
    row++;
  }
  return binPart(mat, M, N, target, mid+1, u, r, row-1, targetRow, targetCol) ||
         binPart(mat, M, N, target, l, row, mid-1, d, targetRow, targetCol);
}

bool binPart(int mat[][N_MAX], int N, int target, int &row, int &col) {
  return binPart(mat, N, N, target, 0, 0, N-1, N-1, row, col);
}
```

Improved Binary Partition:
Since the partition column (or row, or diagonal) is sorted, not utilizing the sorted configuration is a waste. In fact, we are able to modify binary search to search for the partition point in lg *n* time. Then, the complexity can be expressed as the following recurrence relation: (Note: I've omitted some steps, try to work out the math by yourself)

$$
\begin{aligned}
T(n) &= 2T(n/2) + c \lg n \\
&= 2^{\lg n}\, T(1) + c \sum_{k=1}^{\lg n} (2 \lg (n/2^k)) \\
&= O(n) + c\,(2n - \lg n - 2) \\
&= O(n)
\end{aligned}
$$

By incorporating binary search, we are able to reduce the complexity to *O*(*n*). However, we have made an assumption, that is: *Each subdivision of matrices must be of equal size (or, each partition point is exactly at the center of the partition column)*. This leads to the following question:

It is entirely possible that the subdivided matrices are of different sizes. Would the complexity change by an order in this case?

This turns out to be a difficult question to answer, but I could provide further insight to you by deriving the complexity of the other extreme, that is:

Each subdivision results in only one sub-matrix (ie, one matrix has the original matrix being halved, while the other matrix is empty.)

For an example of the above case, try searching for **–1** in the above sample matrix shown in image (*a*). Since each subdivision results in the original matrix being halved, the total number of subdivisions can be at most lg *n* times. Assuming each binary search performed before a subdivision takes *c* lg *n* time, the complexity can be expressed as follow:

```
T(n) = c lg n + ... + c lg n    (a total of lg n times)
     = c lg n * lg n
     = O(lg n)^2
```

As you can see, the run time complexity of this extreme case is $O(\lg n)^2$, which turns out to be even less than $O(n)$. We conclude that this is not the worst case scenario, as some people might believe.

Please note that the worst case for the *Improved Binary Partition* method had not been proven here. We had merely proven that one case of the *Improved Binary Partition* could run in $O(n)$. If you know the proof of the worst case, I would be interested to hear from you.

» **Continue reading Part III: Searching a 2D Sorted Matrix.**

Rating: 4.9/**5** (35 votes cast)

Searching a 2D Sorted Matrix Part II, 4.9 out of 5 based on 35 ratings

← Searching a 2D Sorted Matrix Part I   Searching a 2D Sorted Matrix Part III →

34 thoughts on “Searching a 2D Sorted Matrix Part II”

Chimpanzee
November 10, 2010 at 3:13 pm

Hi buddy,

In your extreme case analysis for improved binary partition, you used '-1' as an example. However, it seems that binary search -1 on diagonal will immediately tell you that -1 is not in the matrix.

How come you can get "one matrix has the original matrix being halved, while the other matrix is empty"?