Searching a 2D Sorted Matrix Part I

October 6, 2010 by 1337c0d3r 21 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,

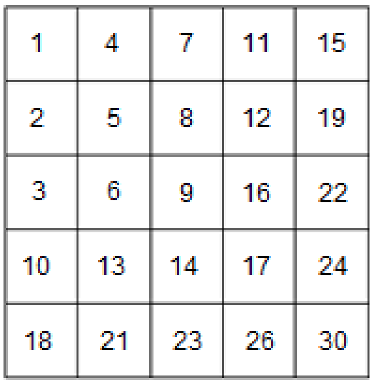
Table[i][j] ≤ Table[i][j + 1],
Table[i][j] ≤ Table[i + 1][j]

Google this problem and you will see that a lot of sites have posted this problem before me. So why another blog post?

I have read through forums, blog posts, and go through each comment people made. Surprisingly, most people had answered this question incorrectly. Especially the run time complexity part. They just put down the recurrence formula and "guess" the complexity. It just don't work like that. To save you from frustration, I decided to post all possible solutions and my in-depth analysis here.

The most efficient solution is surprisingly easy to code. Its run time analysis is also easy to prove. It is also surprisingly easy to make incorrect assumptions if you are not careful.

Note:
For the sake of run time complexity analysis, we will assume that the matrix is a square matrix of size *n* x *n*. Your algorithm should work on matrix of size *m* x *n* though.



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

Elements are sorted across each row and each column in this 2D matrix. This is also known as a Young Tableau.

Hint:

If you are choosing a place to start from, try the top right corner. Or the bottom left corner.

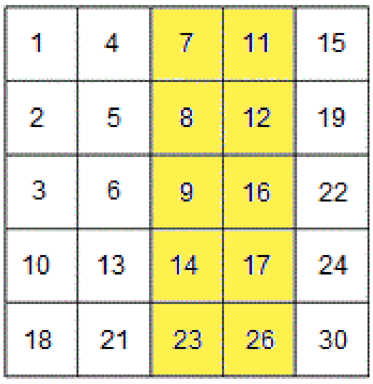
Binary Search Solution:

Most people will observe the keyword "sorted", which no surprise that the idea of applying Binary Search to each row (or to each column) came up immediately. As each row takes *O*(lg *n*) time to search, and there are a total of *n* rows, we are able to do it in *O*(*n* lg *n*) time. Of course, your interviewer will tell you that this is not a good enough solution. In fact, this is the first trap that you should look out for! Do not confine yourself into thinking that binary search is the only way to solve this problem.

Incorrect Solution:

Here is another trap that most people will fall into if they are not careful enough. This is based on an incorrect assumption that if the target element falls between two numbers in the first row, then the target element must be within that two column. If not, then the target element does not exist.

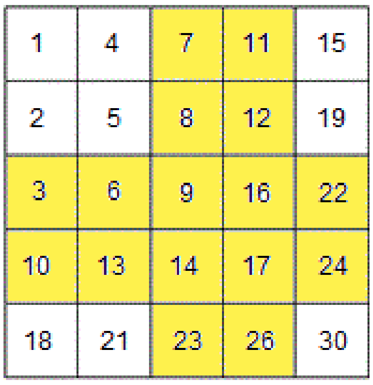
Assume that you are searching for **9**. You scan through the first row and see that **9** appears between **7** and **11**. Therefore, apply Binary Search on these two columns (highlighted in yellow below). This is wrong, try to look for **10** in the matrix. **10** does not belong to any of those two columns.



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

Apply binary search across the highlighted columns. Searches **9** but failed to search for **10**.

What if we do the same to the rows too? That is, since **10** is between **3** and **10**, we apply binary search both to the two rows and two columns (as highlighted in yellow below). Finally, **10** could be found now. Although it seems to work in this example matrix, this is an incorrect solution. Coming up with a counter example is left as an exercise to the reader.



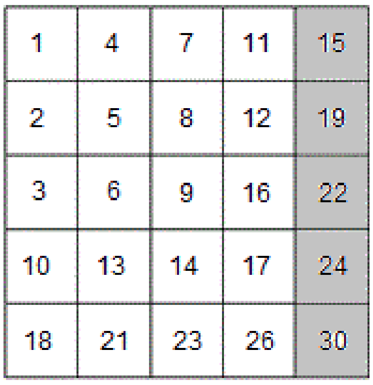
1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

Apply binary search across the highlighted rows and columns. Searches both **9** and **10** now. Is this solution correct? Try to come up with a counter example.

Diagonal Binary Search (Still not there yet):
Still not giving up, you keep on brainstorming on how you can do a Binary Search in a more efficient manner.

How about we traverse the matrix diagonally starting from the top right corner? We will call this method the *Diagonal Binary Search* method. You may also start from the bottom left corner, it doesn't matter.

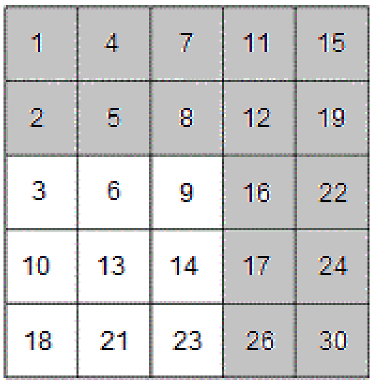
Assume that our target is '10'. We start from the top right corner, which contains the element **15**. Since **10** is less than **15**, we could eliminate the entire column from consideration as illustrated below (Areas that are eliminated are colored in gray). This follows from the rule that the column is in sorted order.



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

Eliminated areas are shown in gray. We start from the upper right corner, and traverse in a diagonal pattern.

We apply binary search from element **1** to **11** on the first row. Since **10** is not found, we proceed to the next diagonal step, which is **12**. **10** is less than **12**, therefore we do a binary search from **2** to **8** across the second row. Since **8** is less than **12**, you could skip doing a binary search anyway. Now, the eliminated areas are shown below. You could see that we continue next with element **9** and do a binary search across the third column from **14** to **23**. Eventually we will find **10** following that.



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

Eliminated areas are shown in gray. We have just completed the second step.

As we traverse diagonally, there's one less item to search for either across the row or column. Therefore, the run time complexity can be written as:

```
T(n) = lg n + lg (n-1) + lg (n-2) + ... + lg (1)
     = lg n(n-1)(n-2)...(1)
     = lg n!
```

Compared to the previous method, this is a tiny bit of improvement over the last method. Proof of lg *n*! is less than *n* lg *n* is left as an exercise to the reader.

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

Vote Saved. Rating: 4.6/**5**

Searching a 2D Sorted Matrix Part I, 4.5 out of 5 based on 15 ratings

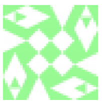
← Excel Sheet Row Numbers Searching a 2D Sorted Matrix Part II →

21 thoughts on "Searching a 2D Sorted Matrix Part I"

rajiv
March 14, 2011 at 12:14 am

http://geeksforgeeks.org/?p=11337

Reply ↓ +1



Adrian M
April 30, 2011 at 8:47 pm

Hi, I like your solution!
I am wondering if it is possible to have another solution

My impression is that the diagonal (top-left to bottom-right) elements are greater than the entire submatrix (which starts at (0,0) and ends at that diagonal element) they lie in.

Say you're looking for element x