<https://www.geeksforgeeks.org/search-element-sorted-matrix/>

# Search element in a sorted matrix

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 25 Jul, 2022

 Read

 Discuss

Given a sorted matrix mat[n][m] and an element ‘x’. Find the position of x in the matrix if it is present, else print -1. Matrix is sorted in a way such that all elements in a row are sorted in increasing order and for row ‘i’, where 1 <= i <= n-1, the first element of row ‘i’ is greater than or equal to the last element of row ‘i-1’. The approach should have O(log n + log m) time complexity.

**Examples:**

Input : mat[][] = { {1, 5, 9},

{14, 20, 21},

{30, 34, 43} }

x = 14

Output : Found at (1, 0)

Input : mat[][] = { {1, 5, 9, 11},

{14, 20, 21, 26},

{30, 34, 43, 50} }

x = 42

Output : -1

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

Please note that this problem is different from [Search in a row wise and column wise sorted matrix](https://www.geeksforgeeks.org/search-in-row-wise-and-column-wise-sorted-matrix/). Here matrix is more strictly sorted as the first element of a row is greater than the last element of the previous row.

A **Simple Solution** is to one by one compare x with every element of the matrix. If matches, then return position. If we reach the end, return -1. The time complexity of this solution is O(n x m).

An **efficient solution** is to typecast a given 2D array to a 1D array, then apply [binary search](https://www.geeksforgeeks.org/binary-search/) on the typecasted array but will require extra space to store this array.

**Another efficient approach** that doesn’t require typecasting is explained below.

1) Perform binary search on the middle column

till only two elements are left or till the

middle element of some row in the search is

the required element 'x'. This search is done

to skip the rows that are not required

2) The two left elements must be adjacent. Consider

the rows of two elements and do following

a) check whether the element 'x' equals to the

middle element of any one of the 2 rows

b) otherwise according to the value of the

element 'x' check whether it is present in

the 1st half of 1st row, 2nd half of 1st row,

1st half of 2nd row or 2nd half of 2nd row.

Note: This approach works for the matrix n x m

where 2 <= n. The algorithm can be modified

for matrix 1 x m, we just need to check whether

2nd row exists or not

**Example:**

Consider: | 1 **2** 3 4|

x = 3, mat = | 5 **6** 7 8| Middle column:

| 9 **10** 11 12| = {2, 6, 10, 14}

|13 **14** 15 16| perform binary search on them

since, x < 6, discard the

last 2 rows as 'a' will

not lie in them(sorted matrix)

Now, only two rows are left

| 1 **2** 3 4|

x = 3, mat = | 5 **6** 7 8| Check whether element is present

on the middle elements of these

rows = {2, 6}

x != 2 or 6

If not, consider the four sub-parts

**1st half of 1st row** = {1}, **2nd half of 1st row** = {3, 4}

**1st half of 2nd row** = {5}, **2nd half of 2nd row** = {7, 8}

According the value of **'x'** it will be searched in the

2nd half of 1st row = **{3, 4}** and found at **(i, j):** (0, 2)

|  |
| --- |
| // java implementation to search  // an element in a sorted matrix  import java.io.\*;    class GFG  {      static int MAX = 100;        // This function does Binary search for x in i-th      // row. It does the search from mat[i][j\_low] to      // mat[i][j\_high]      static void binarySearch(int mat[][], int i, int j\_low,                                      int j\_high, int x)      {          while (j\_low <= j\_high)          {              int j\_mid = (j\_low + j\_high) / 2;                // Element found              if (mat[i][j\_mid] == x)              {                  System.out.println ( "Found at (" + i                                       + ", " + j\_mid +")");                  return;              }                else if (mat[i][j\_mid] > x)                  j\_high = j\_mid - 1;                else                  j\_low = j\_mid + 1;          }            // element not found          System.out.println ( "Element no found");      }        // Function to perform binary search on the mid      // values of row to get the desired pair of rows      // where the element can be found      static void sortedMatrixSearch(int mat[][], int n,                                           int m, int x)      {          // Single row matrix          if (n == 1)          {              binarySearch(mat, 0, 0, m - 1, x);              return;          }            // Do binary search in middle column.          // Condition to terminate the loop when the          // 2 desired rows are found          int i\_low = 0;          int i\_high = n - 1;          int j\_mid = m / 2;          while ((i\_low + 1) < i\_high)          {              int i\_mid = (i\_low + i\_high) / 2;                // element found              if (mat[i\_mid][j\_mid] == x)              {                  System.out.println ( "Found at (" + i\_mid +", "                                      + j\_mid +")");                  return;              }                else if (mat[i\_mid][j\_mid] > x)                  i\_high = i\_mid;                else                  i\_low = i\_mid;          }            // If element is present on          // the mid of the two rows          if (mat[i\_low][j\_mid] == x)              System.out.println ( "Found at (" + i\_low + ","                                   + j\_mid +")");          else if (mat[i\_low + 1][j\_mid] == x)              System.out.println ( "Found at (" + (i\_low + 1)                                  + ", " + j\_mid +")");            // search element on 1st half of 1st row          else if (x <= mat[i\_low][j\_mid - 1])              binarySearch(mat, i\_low, 0, j\_mid - 1, x);            // Search element on 2nd half of 1st row          else if (x >= mat[i\_low][j\_mid + 1] &&                   x <= mat[i\_low][m - 1])          binarySearch(mat, i\_low, j\_mid + 1, m - 1, x);            // Search element on 1st half of 2nd row          else if (x <= mat[i\_low + 1][j\_mid - 1])              binarySearch(mat, i\_low + 1, 0, j\_mid - 1, x);            // search element on 2nd half of 2nd row          else              binarySearch(mat, i\_low + 1, j\_mid + 1, m - 1, x);      }        // Driver program      public static void main (String[] args)      {          int n = 4, m = 5, x = 8;          int mat[][] = {{0, 6, 8, 9, 11},                         {20, 22, 28, 29, 31},                         {36, 38, 50, 61, 63},                         {64, 66, 100, 122, 128}};            sortedMatrixSearch(mat, n, m, x);        }  }    // This code is contributed by vt\_m |

**Output**

Found at (0,2)

Time complexity: O(log n + log m). O(Log n) time is required to find the two desired rows. Then O(Log m) time is required for binary search in one of the four parts with size equal to m/2.

This method is contributed by **Ayush Jauhari**.

**Method 2:** Using binary search in 2 dimensions

This method also has the same time complexity: O(log(m) + log(n)) and auxiliary space: O(1), but the algorithm is much easier and the code way cleaner to understand.

**Approach:** We can observe that any number (say k) that we want to find, must exist within a row, including the first and last elements of the row (if it exists at all). So we first find the row in which k must lie using binary search ( O(logn) ) and then use binary search again to search in that row( O(logm) ).

**Algorithm:**

1) first we’ll find the correct row, where k=2 might exist. To do this we will simultaneously apply binary search on the first and last column.

    low=0, high=n-1

     i) if( k< first element of row(a[mid][0]) ) => k must exist in the row above

                                              => *high=mid-1*;

     ii) if( k> last element of row(a[mid][m-1])) => k must exist in the row below

                                               => *low=mid+1*;

     iii) if( k> first element of row(a[mid][0]) &&  k< last element of row(a[mid][m-1]))

                                               => k must exist in this row

                                               => apply binary search in this row like in a 1-D array

     iv) i) if( k== first element of row(a[mid][0]) ||  k== last element of row(a[mid][m-1])) => *found*

**Example:**

let k=2; n=3,m=4;

matrix a: [0, 1, 2, 3 ]

[10,11,12,13]

[20,21,22,23]

1) low=0, high=n-1(=2) => mid=1 //check 1st row [0....3]

-->[10...13]<--

[20...23]

k < a[mid][0] => high = mid-1;(=1)

2) low=0, high=1; =>mid=0; //check 0th row -->[0...3]<--

k>a[mid][0] && k<a[mid][m-1] => k must exist in this row

now simply apply binary search in 1-D array: [0,1,2,3]

Below is the implementation of the above algorithm:

|  |
| --- |
| // Java program for the above approach  import java.util.\*;  public class Main {        static void findRow(int[][] a, int n, int m, int k)      {          int l = 0, r = n - 1, mid;            while (l <= r) {              mid = (l + r) / 2;                // we'll check the left and              // right most elements              // of the row here itself              // for efficiency              if (k == a[mid][0]) // checking leftmost element              {                  System.out.println("Found at (" + mid + ","                                     + "0)");                  return;              }                if (k == a[mid][m - 1]) // checking rightmost                                      // element              {                  int t = m - 1;                  System.out.println("Found at (" + mid + ","                                     + t + ")");                  return;              }                if (k > a[mid][0]                  && k < a[mid]                          [m - 1]) // this means the element                                   // must be within this row              {                  binarySearch(a, n, m, k,                               mid); // we'll apply binary                                     // search on this row                  return;              }                if (k < a[mid][0])                  r = mid - 1;              if (k > a[mid][m - 1])                  l = mid + 1;          }      }        static void binarySearch(int[][] a, int n, int m, int k,                               int x) // x is the row number      {          // now we simply have to apply binary search as we          // did in a 1-D array, for the elements in row          // number          // x            int l = 0, r = m - 1, mid;          while (l <= r) {              mid = (l + r) / 2;                if (a[x][mid] == k) {                  System.out.println("Found at (" + x + ","                                     + mid + ")");                  return;              }                if (a[x][mid] > k)                  r = mid - 1;              if (a[x][mid] < k)                  l = mid + 1;          }          System.out.println("Element not found");      }        // Driver Code      public static void main(String args[])      {          int n = 4; // no. of rows          int m = 5; // no. of columns            int a[][] = { { 0, 6, 8, 9, 11 },                        { 20, 22, 28, 29, 31 },                        { 36, 38, 50, 61, 63 },                        { 64, 66, 100, 122, 128 } };            int k = 31; // element to search            findRow(a, n, m, k);      }  } |

**Output**

Found at (1,4)