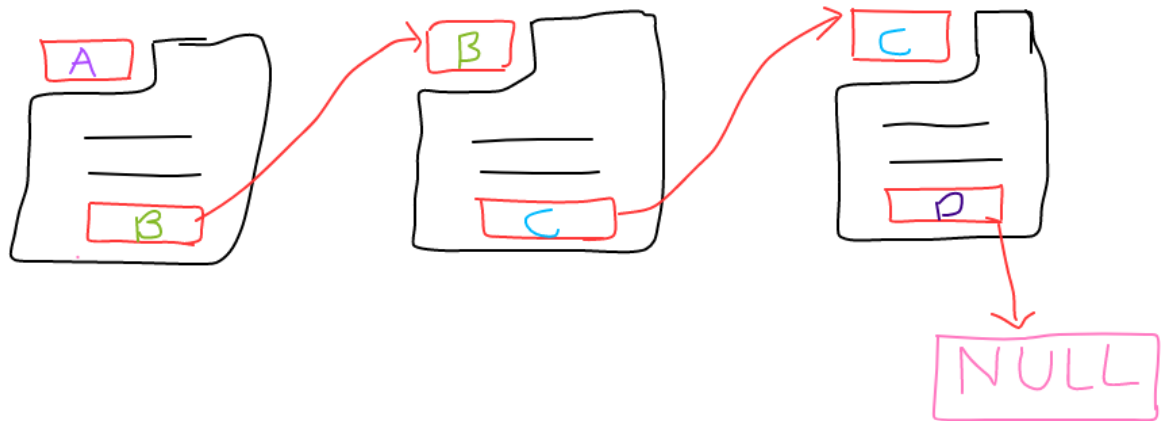


Creating data structures

How to create Linked list using java,



How would structure like shown above would be can be created using java, or consider object oriented programming, we need a concept of address pointing here.

In C, we could do it simply by using pointers.

In java we don't have this address pointer, but we do have the objects. These objects provide the necessary abstraction, which hides the concept of address pointer.

So simply we could use an object type that we specify uniquely, and that's a class.

All you need is class having, property of that class itself, and this object can now store the next class object and this chain can be carry forward for any

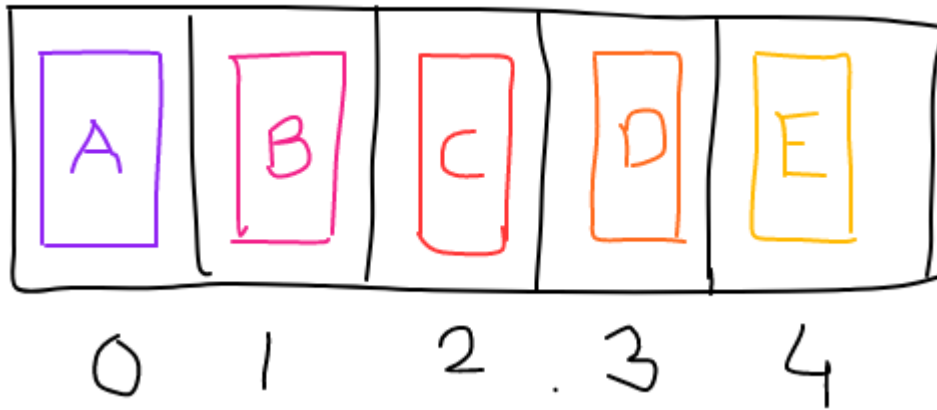
number of times. Also you have creative freedom now so, you could connect many different classes having variety of structures defined by you, and create a data structure of your own.

Finally we need to know, where this chain is going to start and end.

Start can be simply the first object (better idea is to store it another object as head of the list), and let the last object point to null (or store it as end), to iterate through list, this idea will be useful.

This way a singly connected linked list can be created, but the idea now should be intuitive to be extended into doubly connected, circular and any other form of list you like.

Now let's try to implement queue, keeping in mind the first in first out (FIFO) approach.



We are reusing the same objects with sequence as above. Now a queue also keeps tracks of the indexes, so one way to add that functionality is using an array.

There are three main methods that needs to be implemented.

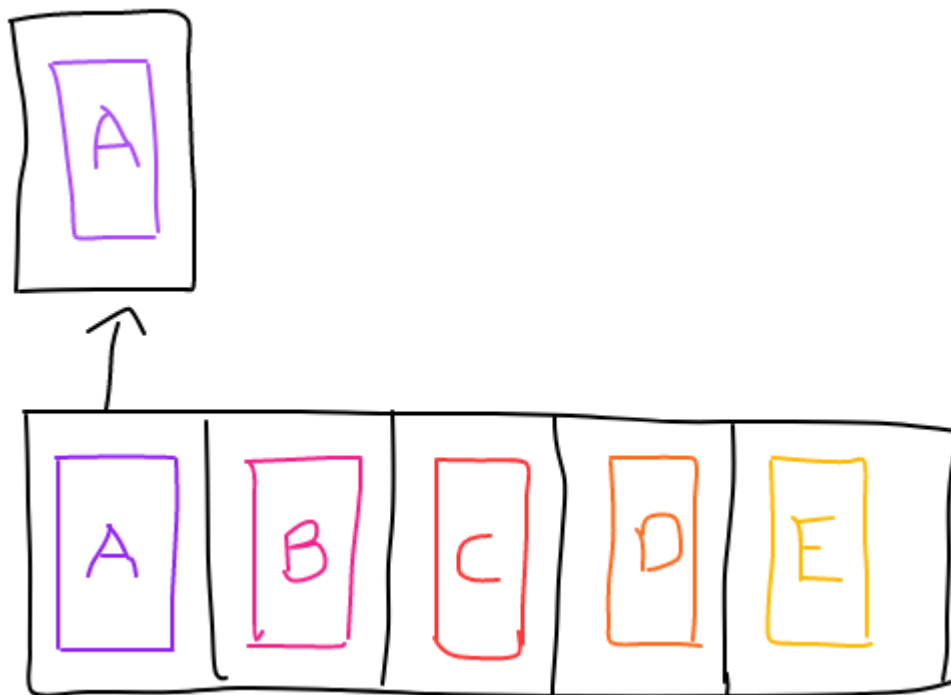
1. peek()
2. poll()
3. add()

Also two complementary methods, isEmpty() and printQueue() would be useful too.

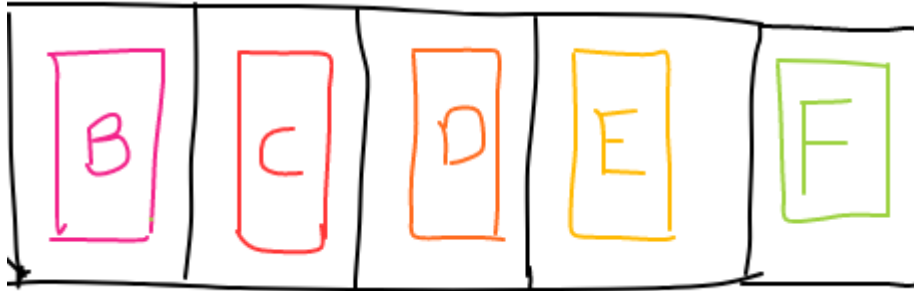
Now begin with isEmpty(), it is confusing sometimes, but we need to just check if an element exists or not, so we could just check the array[0] at initial position, if that is not null we return false; else true.

Next with printQueue(), start by calling isEmpty() method, then simply printing array in the given order, and doesn't break our definition of queue.

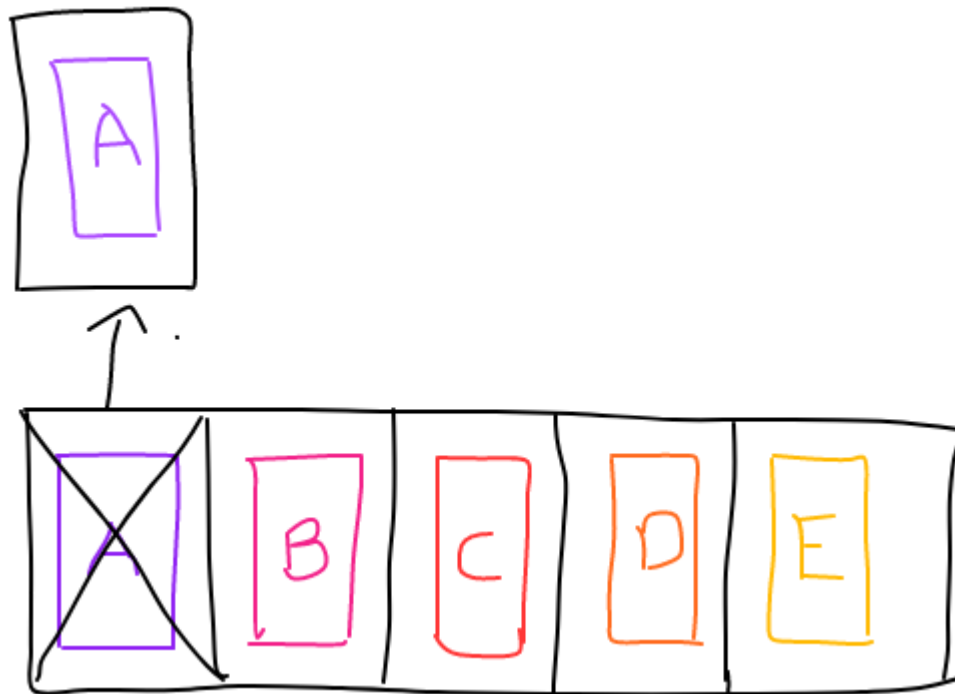
Next for peek(), call isEmpty() and return the first element, since it is FIFO.



For add() method, we will need to resize the array to current array + 1 size , and then add the element at last position. (in code we loop through and copy it one by one, but you could use different ways to achieve the same.)

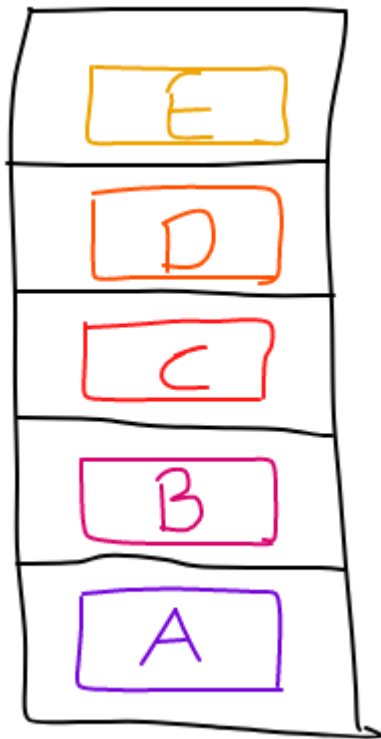


For `poll()` method (removing an element, while returning it), for this again the idea of resizing will be helpful, but only thing to look out for is removing element from the start, and making sure to have a copy of it, as that needs to be returned after resizing array (we use `Arrays` class for this in the code).



Implementing a stack, the approach here is last in first out (LIFO), so think of queue in reverse order, that may clear things up.

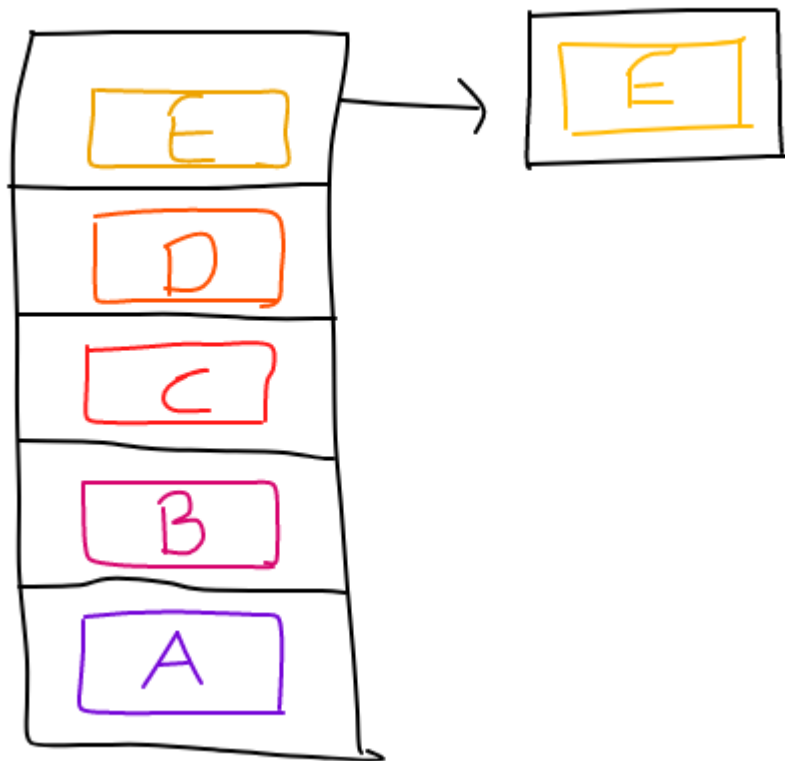
The methods here are isEmpty(), printStack(), pop(), push() and peek().



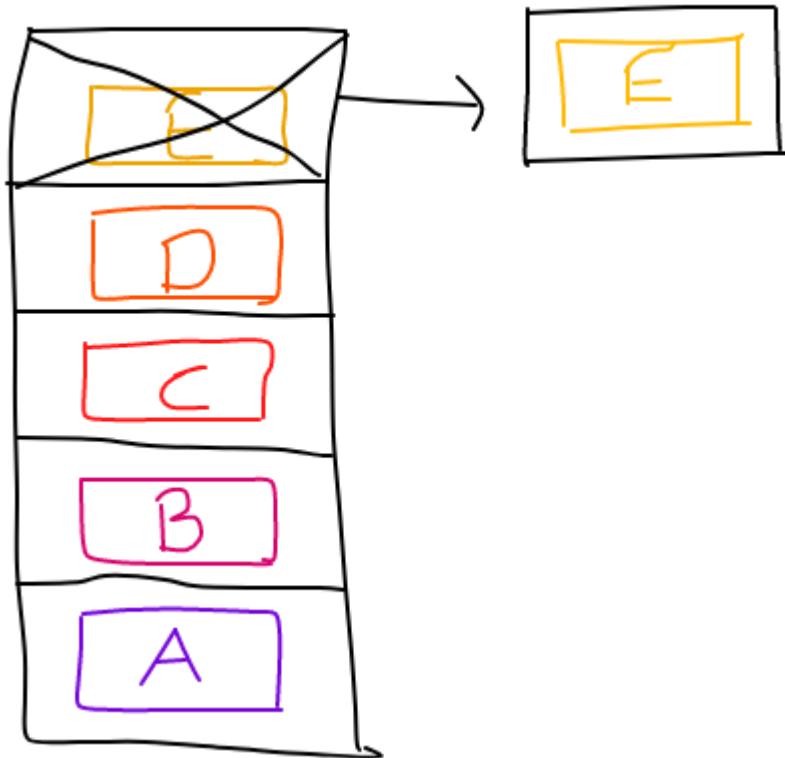
Since we will be using an array here too, for isEmpty() method either the same logic can be used, or to mimic stack, we will check if last element exists or not.

For `printStack()` method, we simply print array in reverse order.

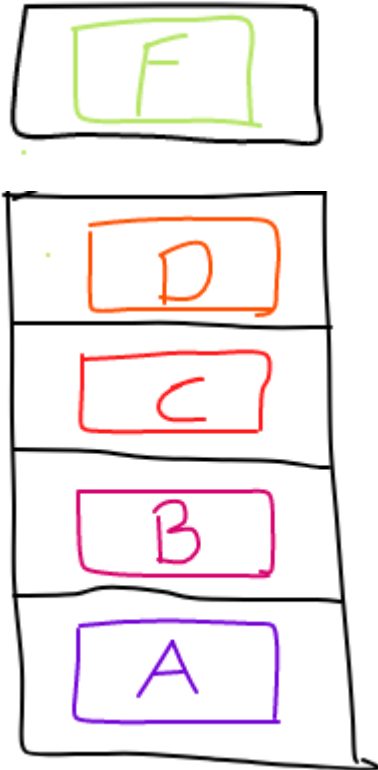
For `peek()`, again we call `isEmpty()` and return the last element.



For `pop()`, we will be returning and removing last element from array and then resizing it.



For push(), we simply resize array and add it at last, as we are reading it in reverse order the method add() has the same use here.



Find numbers that represent binary patterns, upto given n^{th} number.

So if $n = 7$,

return

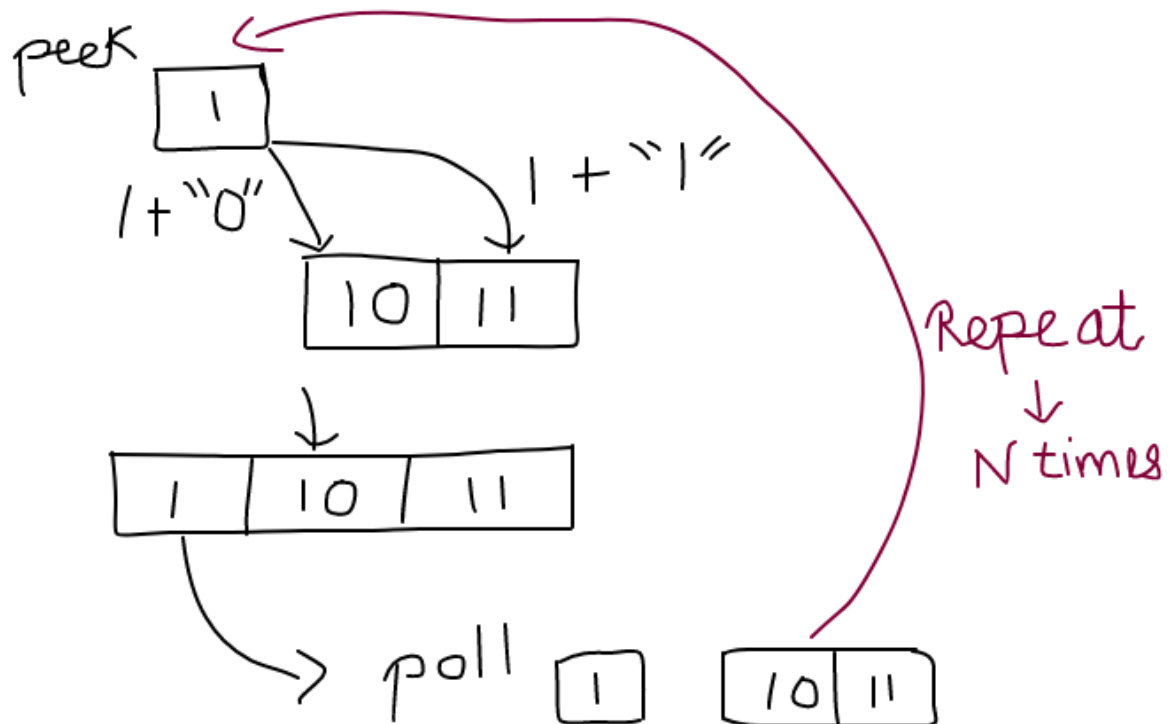
|
| 0
| 1
| 0 0
| 0 1
| 1 0
| 1 1

Now seeing these numbers as string , we could break it into easier counting problem.

Here a pattern can be observed, where every next two numbers ends with 0 and 1, which are appended to current number (in our case its 1), and next numbers are 10 and 11.

And this continues for next two 100 and 101 (for 10).

A way to solve this is queue as depicted below,



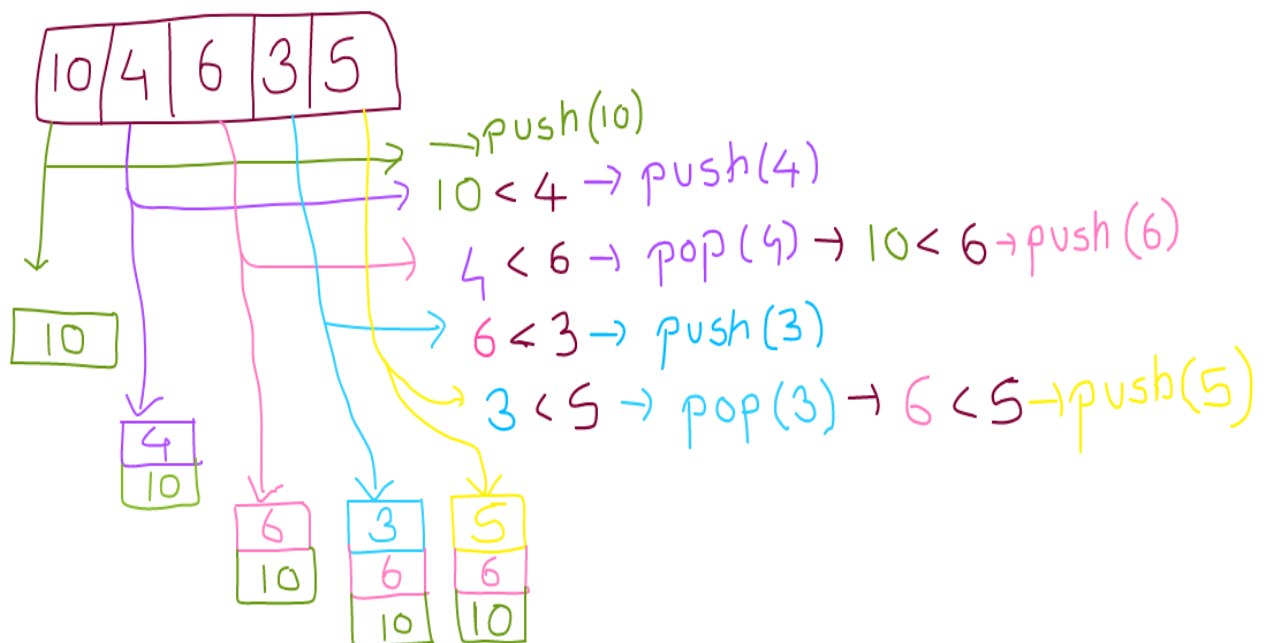
Here we first add "1" to queue, and then start looping through n times. We first concatenate "0" and "1", with the starting element of queue ("1" in this case).

Finally poll the element and print it, this process is repeat until we print n elements.

Given an array of n numbers { 10 , 4 , 6 , 3 , 5 }, return the numbers which have all numbers smaller to right of it.

So we return 10 , 6 , 5 , in this case, the ordering of numbers returned does not matter.

A way to solve this using stack, is depicted below.



Here we add to stack each element, while we are iterating through the array.

We check if the element on top of the stack is smaller than the current element.

If that's the case, we pop that element.

Now here you could think, the stack is being filled with only element which is greater than current element, that is the exact problem statement, but we are checking it in reverse order.

Finally once iteration of array is over, only the elements which are maximum from all the elements to its right remains, so we print them.

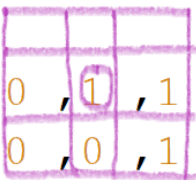
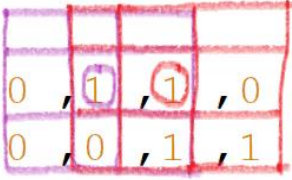
Connected Cells

Here we want to solve the problem of finding maximum number of 1's connected across all direction in given matrix.

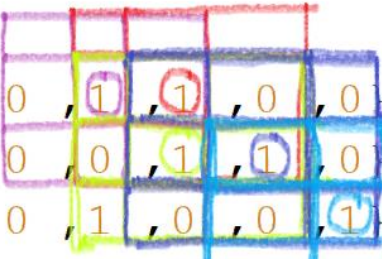

(7 in given case)

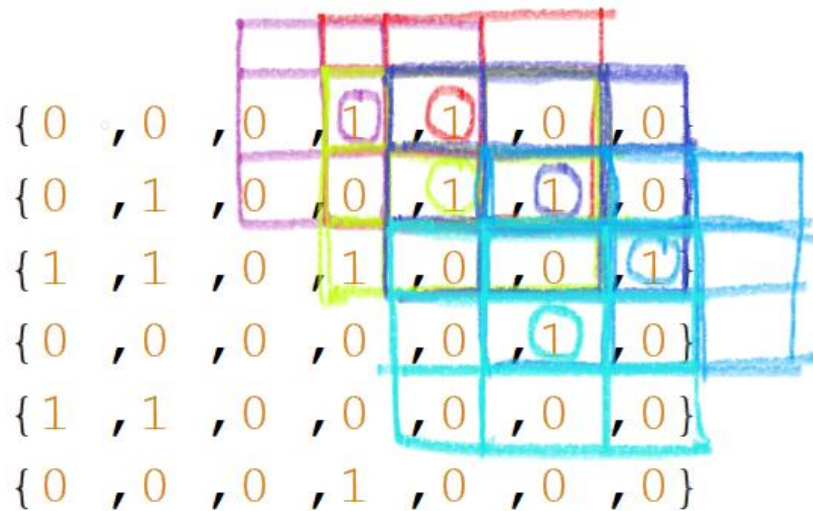


We solve this problem using the Depth First Search (DFS) approach.

	
{ 0 , 0 , 0 , 1 , 1 , 0 , 0 }	{ 0 , 0 , 0 , 1 , 1 , 0 , 0 }
{ 0 , 1 , 0 , 0 , 1 , 1 , 0 }	{ 0 , 1 , 0 , 0 , 1 , 1 , 0 }
{ 1 , 1 , 0 , 1 , 0 , 0 , 1 }	{ 1 , 1 , 0 , 1 , 0 , 0 , 1 }
{ 0 , 0 , 0 , 0 , 0 , 1 , 0 }	{ 0 , 0 , 0 , 0 , 0 , 1 , 0 }
{ 1 , 1 , 0 , 0 , 0 , 0 , 0 }	{ 1 , 1 , 0 , 0 , 0 , 0 , 0 }
{ 0 , 0 , 0 , 1 , 0 , 0 , 0 }	{ 0 , 0 , 0 , 1 , 0 , 0 , 0 }

	
{ 0 , 0 , 0 , 1 , 1 , 0 , 0 }	{ 0 , 0 , 0 , 1 , 1 , 0 , 0 }
{ 0 , 1 , 0 , 0 , 1 , 1 , 0 }	{ 0 , 1 , 0 , 0 , 1 , 1 , 0 }
{ 1 , 1 , 0 , 1 , 0 , 0 , 1 }	{ 1 , 1 , 0 , 1 , 0 , 0 , 1 }
{ 0 , 0 , 0 , 0 , 0 , 1 , 0 }	{ 0 , 0 , 0 , 0 , 0 , 1 , 0 }
{ 1 , 1 , 0 , 0 , 0 , 0 , 0 }	{ 1 , 1 , 0 , 0 , 0 , 0 , 0 }
{ 0 , 0 , 0 , 1 , 0 , 0 , 0 }	{ 0 , 0 , 0 , 1 , 0 , 0 , 0 }

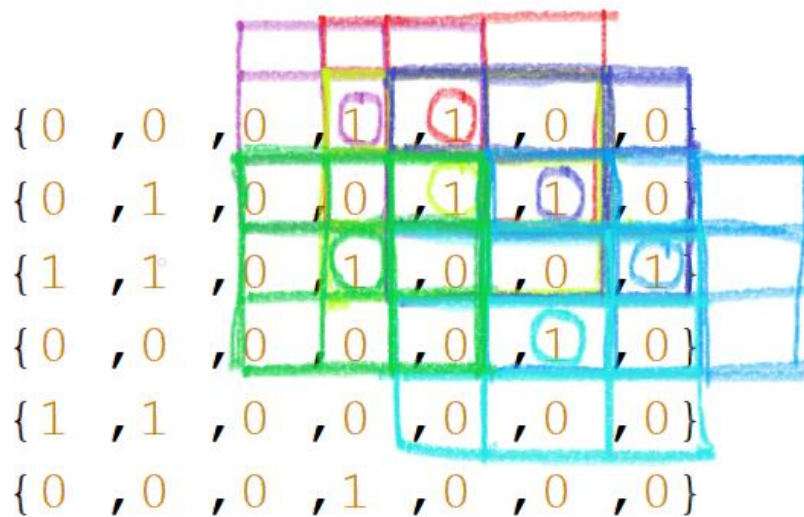
	
{ 0 , 0 , 0 , 1 , 1 , 0 , 0 }	{ 0 , 0 , 0 , 1 , 1 , 0 , 0 }
{ 0 , 1 , 0 , 0 , 1 , 1 , 0 }	{ 0 , 1 , 0 , 0 , 1 , 1 , 0 }
{ 1 , 1 , 0 , 1 , 0 , 0 , 1 }	{ 1 , 1 , 0 , 1 , 0 , 0 , 1 }
{ 0 , 0 , 0 , 0 , 0 , 1 , 0 }	{ 0 , 0 , 0 , 0 , 0 , 1 , 0 }
{ 1 , 1 , 0 , 0 , 0 , 0 , 0 }	{ 1 , 1 , 0 , 0 , 0 , 0 , 0 }
{ 0 , 0 , 0 , 1 , 0 , 0 , 0 }	{ 0 , 0 , 0 , 1 , 0 , 0 , 0 }



A 6x7 matrix of binary values. The matrix is:

{ 0	, 0	, 0	, 1	, 1	, 0	, 0
{ 0	, 1	, 0	, 0	, 1	, 1	, 0
{ 1	, 1	, 0	, 1	, 0	, 0	, 1
{ 0	, 0	, 0	, 0	, 0	, 1	, 0
{ 1	, 1	, 0	, 0	, 0	, 0	, 0
{ 0	, 0	, 0	, 1	, 0	, 0	, 0

 The matrix is annotated with colored borders and circled 1s. The first row has a red border. The second row has a yellow border. The third row has a green border. The fourth row has a blue border. The fifth row has a cyan border. The sixth row has a magenta border. The 1s in the first row are circled in red. The 1s in the second row are circled in yellow. The 1s in the third row are circled in green. The 1s in the fourth row are circled in blue. The 1s in the fifth row are circled in cyan. The 1s in the sixth row are circled in magenta.



A 6x7 matrix of binary values. The matrix is:

{ 0	, 0	, 0	, 1	, 1	, 0	, 0
{ 0	, 1	, 0	, 0	, 1	, 1	, 0
{ 1	, 1	, 0	, 1	, 0	, 0	, 1
{ 0	, 0	, 0	, 0	, 0	, 1	, 0
{ 1	, 1	, 0	, 0	, 0	, 0	, 0
{ 0	, 0	, 0	, 1	, 0	, 0	, 0

 The matrix is annotated with colored borders and circled 1s. The first row has a red border. The second row has a yellow border. The third row has a green border. The fourth row has a blue border. The fifth row has a cyan border. The sixth row has a magenta border. The 1s in the first row are circled in red. The 1s in the second row are circled in yellow. The 1s in the third row are circled in green. The 1s in the fourth row are circled in blue. The 1s in the fifth row are circled in cyan. The 1s in the sixth row are circled in magenta.

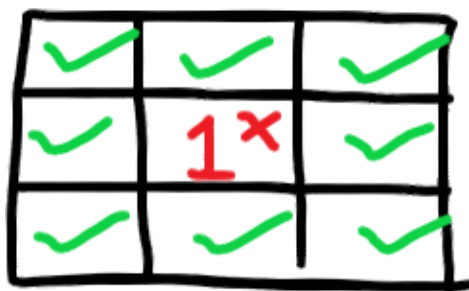
Here we iterate through each element of matrix, but we need to keep a track of maximum length of 1's.

Once 1 is found, we enter the recursive function, where the base case checks the boundaries of the corners and if the current element is 0.

Then we make the current element 0, so that we could just eliminate overlaps. Now as we check the entire row before the current row till after the current row, for all columns before and after the current column, as shown in the color grids above.

This finds a 1, which is not yet traversed, then recursive function is called, and we do the same for it. For all the recursive function which has 1 as current element, we return a 1, that is added up recursively to above function called, this way the track of size is kept.

Notice, that current row and column, element is already switched to 0, so in loops where we traverse the grid mentioned, we eliminate this element and continue for all others, as shown below (the if condition in code).



✓	✓	✓
✓	1 ^x	✓
✓	✓	✓

As we traverse down each time 1 is found, hence this follows DFS solution.